

OFFICIAL

THE

matt tagliaferri

DUKE NUKEM

3D



**THE OFFICIAL LEVEL
DESIGN GUIDE**



**DUKE'S CREATORS
REVEAL THEIR SECRETS**



50 NEW Levels!

**Convert DOOM
Levels into Duke
Adventures!
Exclusive
Conversion
Program
Included.**

LEVEL DESIGN HANDBOOK



**3D
REALMS**

BUILD KEYSTROKES AND OPERATIONS FOR 2D VIEW MODE

| | |
|--|---|
| spacebar | Starts sector drawing mode/create new vertex/create vertex on top of existing vertex in same sector |
| backspace | Delete previous vertex/end sector drawing mode when last vertex |
| Enter (numeric keypad) | Toggle between 2D and 3D view modes |
| Arrow keys | Change viewpoint position (move the white arrow) in 2D or 3D |
| Scroll Lock | Change player one start position to current white arrow location |
| Esc, S | Save a level |
| Esc, L, choose level, Enter | Load a Level |
| Esc, A, type new name, Enter | Save level with a new name |
| Esc, N, Y | Start a new level |
| Esc, Q, Y, Y | Quit Build |
| Esc, Y, Y | Save a level |
| A | Zoom in |
| Z | Zoom out |
| Hold left mouse button and drag | Move a vertex or sprite |
| Ins | Split a wall (insert vertex) |
| Connect two vertices of same sector in sector drawing mode | Split a sector |
| J (with mouse cursor on first sector), J again (with mouse cursor on second sector) | Join a sector |
| Drag vertex onto neighbor vertex | Delete a vertex |
| Alt+S (with mouse inside sector) | Create a child sector |
| Right Shift+drag to enclose vertices in rectangle | Select multiple vertices |
| Drag selection rectangle while holding Right Shift key | Move all vertices at once |
| Right Ctrl+Del | Delete a sector |
| Right Alt+drag to enclose sectors in a rectangle | Select a sector |
| Ins+press and hold left mouse button with mouse cursor over a sector | Copy a sector and drag the copy to a new location |
| , (comma) key | Rotate selected sector or sectors 90 degrees clockwise |
| . (period) key | Rotate selected sector or sectors 90 degrees counterclockwise. |

| | |
|-------------------------|---|
| Shift+, (comma) | Rotate selected sector or sectors clockwise one degree at a time |
| Shift+. (period) | Rotate selected sector or sectors counterclockwise one degree at a time |

BUILD KEYSTROKES AND OPERATIONS FOR 3D VIEW MODE

| | |
|---|--|
| PgUp (mouse cursor on floor/ceiling and left mouse button held down) | Raise floor/ceiling height |
| PgDn (mouse cursor on floor/ceiling and left mouse button held down) | Lower ceiling height |
| V (once), Enter/ V (twice), Enter | Choose a texture from those used in current level/choose a texture from all available textures |
| Tab | Copy a texture or sprite to the clipboard |
| Enter | Paste a texture from the clipboard |
| Ctrl+Enter | Paste the texture in the clipboard on all walls in a loop |
| [or] (mouse cursor on floor/ceiling and left mouse button held down) | Slope a floor or ceiling |
| Alt+F | Change sector's first wall in 2D or 3D |
| 2 | Allow top and bottom of wall to have different textures |
| M | Create a masked wall (both sides) |
| Shift+M | Create a masked wall (one side) |
| Alt+C | Paste wall texture in clipboard onto all walls on level with current texture |
| O | Change painting orientation of wall texture (top-down vs. Bottom-up) |
| 2, 4, 6, and 8 (numeric keypad) | Scale current wall or sprite texture or pan current wall, ceiling, or floor texture |
| / | Reset wall texture scale or panning to default |
| . (period) | Align all like wall textures in a loop automatically |
| E | Scale texture (two possible sizes) |
| R | Relative map ceiling or floor texture |
| P | Parallax ceiling or floor texture |
| Ctrl+P | Change parallax mode for ceiling or floor |
| + (plus), - (minus), or '+S | Shade wall, ceiling, floor, or sprite |
| Shift+Enter | Paste shade value of texture in clipboard onto current wall, ceiling, or floor |

| | |
|---------------------------------|---|
| Alt+P | Change wall, ceiling, or floor palette |
| S | Create a new sprite in 2D or 3D |
| PgUp or PgDn | Move sprite up or down in sector |
| Ctrl+PgUp or Ctrl+PgDn | Align sprite to ceiling or floor |
| R | Change sprite display mode |
| , (comma) and . (period) | Change sprite angle in 2D or 3D |
| O | Move and change sprite to lie flat against nearest wall |
| Del | Delete a sprite |
| F | Flip sprite texture |
| B | Turn on Blocking bit |
| H (3D) and Ctrl+H (2D) | Turn on HitScan bit in 2D or 3D |
| T (three modes) | Make sprite translucent |

EDITART SELECTION AND MANIPULATION KEYS

| | |
|------------------|---|
| U | Import a section of a 320x200x256 BMP, PCX, or GIF graphic |
| Enter | Convert image inside selection rectangle to the BUILD palette |
| Spacebar | Convert image inside selection rectangle without remapping the palette |
| P | If in the picture selecting screen (after pressing U and loading the picture), replace Build's palette with the picture's palette |
| PgUp/PgDn | Select tile to edit |
| G | Go to a tile by typing the tile number |
| S | Re-size a tile |
| Del | Set both the X and Y sizes to 0 |
| +, - | Change the animation setting |
| A | Set the animation speed of the tile |
| ` | Center a sprite |
| N | Name a tile |
| O | Optimize the size of an individual piece of artwork |
| V | View and select a tile to edit |
| spacebar | Swap two tiles (press spacebar on each) |
| 1, 2, 3 | Swap a group of tiles, (press 1 on first, press 2 to remember region between, and press 3 at swap location) |
| Alt+U | Re-grab artwork from original pictures according to the CAPFIL.TXT file |

| | |
|--------------|---|
| Alt+R | Generate a tile frequency report for all maps in directory in View mode |
| F12 | Screen capture (saves image as a BMP file and numbers file names consecutively) |
| Esc | Quit |

EDITART GRAPHICS EDITING AND TOUCH-UP KEYS

| | |
|---------------------|---|
| C | Change all same color pixels on the tile to the selected color |
| Arrows/Mouse | Move graphics cursor |
| Shift+Arrows | Select color (on bottom right corner of screen) |
| Spacebar | Plot a pixel with selected color |
| T | Turn drawing trail on and off |
| Tab | Select the color under the graphics cursor |
| Backspace | Set the color to color 255 (transparent color) |
| F | Flood fill a region with current color and with current color as boundary |
| M/P | Back up a tile into a temporary memory buffer/restore it |
| J | Randomly plots dots of current color over any pixels having the same color as color under tile cursor |
| [| Random anti-alias of colors in color band under graphics cursor |
|] | Non-random anti-alias of colors in color band under graphics cursor |
| ; | Make an image 3D |
| ' | Make an image 3D the other way |
| R | Rotate the tile in a specified direction |
| 1 | Mark first corner of rectangle for copy/paste |
| 2 | Mark other corner of rectangle for copy/paste |
| 3 | Paste the selected rectangle |
| 4 | Flip the copied rectangular region x-wise |
| 5 | Flip the copied rectangular region y-wise |
| 6 | Swap the x and y coordinates of the copied rectangular region |
| ,.<> | Change shade of selected region |
| \ | Move the cursor to center of tile |
| | Get the coordinates of cursor |

The Duke Nukem[®] 3D Level Design Handbook

by matt tagliaferri

3D
REALMS



SAN FRANCISCO

PARIS

DÜSSELDORF

SOEST

| | |
|-------------------------------|-------------------------------|
| Associate Publisher | GARY MASTERS |
| Acquisitions Manager | KRISTINE PLACHY |
| Project Editor | MAUREEN ADAMS |
| Production Coordinator | LABRECQUE PUBLISHING SERVICES |
| Developmental and Copy Editor | TERRENCE O'DONNELL |
| Book Design and Production | WILLIAM SALIT |
| Proofreader | TANYA KUCAK |
| Cover Designer | ARCHER DESIGN |

Wizardry is a registered trademark of Sir-Tech Software.

3D Realms Entertainment is a division of Apogee Software, Ltd.

DOOM, *Hexen*, *Wolfenstein 3D*, and *Heretic* are registered trademarks of id Software, Inc.

CD-ROM map authors: BLDFAQ09.TXT, Steffen "Duke Addict" Itterheim, 100606.2141@compuserve.com; DN3DFAQ.TXT, Klaus Breuer, sz0759@rzmail.uni-erlangen.de; 2ATWAR.map, Daniel ?, 75763.1434; 7thGuest.map, Allan Arico, 103057.2343@compuserve.com; anzag.map, Hans Benz, ANZAG@aol.com; Apocalypse, Mike Greig, 101732.1363@compuserve.com; Arena.map, Chris, chris@slayer.powernet.co.uk; Bartmap3.map, Jim Wiscarson (AKA) Bartman, Bartman@aceinfo.com; CyKoSiS.map, Jason Williams, InSaNe1@Xband.com; Funhouse.map, Jason Williams, InSaNe1@Xband.com; DUKECTRL.map, Scott McNutt, smmcnutt@best.com; DUKEPROX.map, Scott McNutt, smmcnutt@best.com; Fortress.map, Nick Shore, Nick Depree, mlshore@central.co.nz; FORTS.map, Ari Samsky, Cthulhu@worldnet.att.net; HALLHELL.MAP, Gary D. Danielson Jr., Compuserve-(102704.2407); HULK.map, Tom Francis, 100565.572; ICEWORLD.map, Inge Groben, Ingegrogen@aol.com; Surprise, Inge Groben, Ingegrogen@aol.com; Kilo16.map, KiloWatt, kilowatt@earthlink.net; Labrynth.map, Sidhe, 73564.3440@compuserve.com; LVL7.MAP, Alex 'INXS' Linn, 101676.746@compuserve.com; Milbase.map, Sidhe, 73564.3440@compuserve.com; LTEMPLE.map, Scott Haslam, 102761.222; MaNCeR01.MAP, Cole Savage, cole@dodgenet.com; MANSION.map, Cho Yan Wong (Tempest), pwong@pobox.leidenuniv.nl; MOONSHOT.MAP, Richard M. Gold (aka; Phantom), 72734.2553@compuserve.com; PAYBACK.map, Richard M. Gold (aka; Phantom), 72734.2553@compuserve.com; ROQ.map, Cho Yan Wong (Tempest), pwong@pobox.leidenuniv.nl; SMACK.map, Cho Yan Wong (Tempest), pwong@pobox.leidenuniv.nl; Mnymoon.map, James Healey, 101325.1604@compuserve.com; Morphgen.map, James Healey, 101325.1604@compuserve.com; Hood.map, Jonathan E Wildman, 101461.514@Compuserve.com; Mufambi.map, Simon Crawshaw, 100610.3067@compuserve.com; normal.map, Michael Clark, 100660.2373@compuserve.com; Omega Outpost, Phil Smith (Override), 73473.1323@compuserve.com; Pitperil.map, Matthew R. King, 74757.3547@compuserve.com; POSH1.map, Matthew Falcus, 100257.2007@compuserve.com; Resort.map, Nick Shore, mlshore@central.co.nz; RidHouse.map, Robin Ridler, 100711.2025@compuserve.com; Sand.map, Neil Munday (Punisher), Jim Semkiw (Ironman), jsemkiw@direct.ca or nam@direct.ca; Startrek.map, A,Covington and I,Stubbington., alanc@mail.bogo.co.uk; Steel.map, Matt Bollier, 103220.3302@compuserve.com; strng.map, Roland Blais, roland@greatbasin.com; Subcity.map, BowZer, bowser@magi.com; Treptow.map, Peter Scholz, 101607.3643@compuserve.com; TRJX1.map, TRJ, 72643.3336@compuserve.com; XTDN_10! starter program, J_rgen Sommer, 100337.2242.

SYBEX is a registered trademark of SYBEX Inc.

TRADEMARKS: SYBEX has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer.

Every effort has been made to supply complete and accurate information. However, SYBEX assumes no responsibility for its use, nor for any infringement of intellectual property rights of third parties which would result from such use.

Copyright © 1996 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

Library of Congress Card Number: 96-70201

ISBN: 0-7821-1869-0

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Warranty

Sybex warrants the enclosed CD-ROM to be free of physical defects for a period of ninety (90) days after purchase. If you discover a defect in the CD during this warranty period, you can obtain a replacement CD at no charge by sending the defective CD, postage prepaid, with proof of purchase to:

Sybex Inc.
Customer Service Department
1151 Marina Village Parkway
Alameda, CA 94501
(800) 227-2346
Fax: (510) 523-2373

After the 90-day period, you can obtain a replacement CD by sending us the defective CD, proof of purchase, and a check or money order for \$10, payable to Sybex.

Disclaimer

Sybex makes no warranty or representation, either express or implied, with respect to this medium or its contents, its quality, performance, merchantability, or fitness for a particular purpose. In no event will Sybex, its distributors, or dealers be liable for direct, indirect, special, incidental, or consequential damages arising out of the use of or inability to use the software even if advised of the possibility of such damage.

The exclusion of implied warranties is not permitted by some states. Therefore, the above exclusion may not apply to you. This warranty provides you with specific legal rights; there may be other rights that you may have that vary from state to state.

Shareware Distribution

This CD contains various programs that are distributed as shareware. Shareware is a distribution method, not a type of software. The chief advantage is that it gives you, the user, a chance to try a program before you buy it.

Copyright laws apply to both shareware and commercial software, and the copyright holder retains all rights. If you try a shareware program and continue using it, you are expected to register it. Individual programs differ on details—some request registration while others require it. Some request a payment, while others don't, and some specify a maximum trial period. With registration, you get anything from the simple right to continue using the software to program updates.

Copy Protection

None of the files on the CD is copy-protected. However, in all cases, reselling or making copies of these files without authorization is expressly forbidden.

DEDICATION

To my “good lady” wife, Janet, and her infinite patience.

ACKNOWLEDGMENTS

I’d first like to thank the entire 3D Realms team for putting out such a quality piece of entertainment software and taking the 3D first-person shoot-’em-up game genre to a new level. I’d especially like to thank Scott Miller of 3D Realms, who basically got this book-writing gig for me. Other thanks need to go to Richard (Levelord) Bailey Gray, Allen H. Blum III, and Todd Replogle, who really went out of their way to contribute to the book. Next on the list comes Gary Masters, Associate Publisher at Sybex, who took Scott Miller’s recommendation and allowed me the opportunity to write, and Terry O’Donnell, who took what were in the truest sense of the word first drafts and shaped them into readable text. Finally, I’d like to thank all of you, the readers and game players, for simply sharing in one of my greatest loves: the computer game.

TABLE OF CONTENTS

Introduction

xii

CHAPTER 1

The 3D First-Person Shoot-'Em-Up Game Genre

2

| | |
|------------------------------------|----|
| BEFORE DUKE NUKEM WENT 3D | 4 |
| The Doom Legacy | 5 |
| THE DAWNING OF DUKE NUKEM 3D | 6 |
| Multiplayer Ability | 7 |
| Weapons | 7 |
| Monsters | 9 |
| Power-ups | 11 |
| ADDING YOUR OWN TOUCH | 13 |

CHAPTER 2

Introduction to Build

14

| | |
|---|----|
| BUILD'S GOOD POINTS | 16 |
| Viewing the Design in 3D Mode | 16 |
| Designing Operations Is Easy | 18 |
| Working with Build-Provided Examples | 18 |
| BUILD'S LIMITATIONS | 19 |
| Working with Build's Hot Key-based Interface | 19 |
| Working with "Not -So-Friendly" Documentation | 19 |
| Seeing Moving Sectors Is Not Possible | 20 |
| STARTING BUILD | 20 |
| BUILDING YOUR FIRST LEVEL | 22 |
| The Elements of a Sector | 23 |
| Creating a Sector | 23 |
| Studying Actual Game Sectors | 25 |
| Sector Case Study #1: The Saloon | 25 |
| Sector Case Study #2: The Submarine | 28 |
| Sector Case Study #3: The Pagan Chapel | 28 |
| Viewing Your Own Level in 3D | 31 |
| Saving Your Own 3D Level | 33 |
| Keystroke and Operation Summary | 34 |

CHAPTER 3

Creating Sector Groups

36

| | |
|---|----|
| RETURNING TO BUILD | 38 |
| MODIFYING A SECTOR | 39 |
| Zooming Your View | 39 |
| Changing the Shape of a Sector | 39 |
| SPLITTING WALLS | 41 |
| CREATING A NEW SECTOR | 41 |
| Adjusting Floor and Ceiling Heights | 45 |



THE DUKE NUKEM 3D Level Design Handbook

| | |
|---|----|
| SPLITTING ONE SECTOR INTO TWO..... | 48 |
| JOINING TWO SECTORS INTO ONE | 50 |
| Deleting Vertices | 50 |
| CREATING A CHILD SECTOR | 51 |
| SELECTING TEXTURES..... | 55 |
| CREATING OTHER SECTOR TYPES | 57 |
| Drawing the Peninsula Sector | 57 |
| Drawing the Corner Sector | 58 |
| MOVING A SECTOR | 59 |
| DELETING A SECTOR..... | 59 |
| COPYING A SECTOR | 61 |
| Copying Child Sectors | 63 |
| ROTATING A SECTOR | 65 |
| SLOPING FLOORS AND CEILINGS | 66 |
| Controlling the Direction of Slope | 66 |
| CREATING SECTORS OVER SECTORS | 67 |
| DO SOME ARCHITECTURAL INVESTIGATION | 69 |
| Keystroke and Operation Summary | 69 |

CHAPTER 4

Bringing Sectors to Life 72

| | |
|--|----|
| EMBELLISHING WALLS | 74 |
| Changing Wall Textures | 75 |
| Using Different Textures on Top and Bottom Wall Sections | 75 |
| Creating Masked Walls | 76 |
| Copying and Pasting Textures | 77 |
| Replacing Textures in an Entire Level | 78 |
| Controlling the Painting of Textures..... | 78 |
| Scaling Wall Textures | 79 |
| Panning Wall Textures | 81 |
| Aligning Wall Textures | 82 |
| EMBELLISHING CEILINGS AND FLOORS | 83 |
| Changing Ceiling and Floor Textures | 83 |
| Copying Textures for Ceilings and Floors | 83 |
| Scaling Ceiling and Floor Textures..... | 83 |
| Panning Ceiling and Floor Textures | 85 |
| Relative Mapping | 86 |
| Parallaxing Textures | 88 |
| APPLYING SHADE | 89 |
| APPLYING PALETTE | 91 |
| EXAMPLE APPLICATIONS OF TEXTURE, LIGHT, AND PALETTE..... | 92 |
| Case Study #1: The Dark Side (E2L8)—Courtyard Area..... | 92 |
| Case Study #2: The Incubator (E2L2)—Blue Area..... | 95 |
| Case Study #3: The Dark Side (E2L8)—Monolith Area | 96 |
| One Final Word on Sector Design | 97 |
| Keystroke and Operation Summary | 98 |

CHAPTER 5

Placing Objects with Sprites 100

| | |
|------------------------------------|-----|
| WORKING WITH SPRITES | 102 |
| Creating Sprites | 103 |
| Moving Sprites | 103 |
| Applying Textures to Sprites | 103 |



| | |
|---------------------------------------|-----|
| Selecting Monster Textures | 104 |
| SPRITE DISPLAY VIEWS | 105 |
| Setting Sprite Display Modes | 106 |
| Sprite Viewpoint Angles | 108 |
| Sizing Sprites..... | 110 |
| Shading Sprites | 110 |
| MANIPULATING SPRITES | 111 |
| Copying Sprites | 111 |
| Deleting Sprites | 111 |
| Flipping Sprites | 111 |
| Blocking Sprites | 112 |
| Creating Translucent Sprites | 113 |
| Keystroke and Operation Summary | 114 |

CHAPTER 6

Special Types of Sprites 116

| | |
|--|-----|
| CREATING EFFECTS WITH LOTAG AND HITAG VALUES | 118 |
| Linking Sprites with Unique Values | 119 |
| Setting LoTag and HiTag Values | 119 |
| The SectorEffector Sprite (1) | 120 |
| The Activator Sprite (2) | 121 |
| The TouchPlate Sprite (3)..... | 121 |
| The ActivatorLocked Sprite (4) | 122 |
| The Music&SFX Sprite (5) | 123 |
| The Locator Sprite (6) | 124 |
| The Cycler Sprite (7) | 124 |
| The MasterSwitch Sprite (8) | 124 |
| The Respawn Sprite (9) | 125 |
| The GPSpeed Sprite (10) | 126 |
| The AccessSwitch Sprite (130) | 126 |
| The Switch Sprite | 126 |
| The NukeButton Sprite (142) | 127 |
| The MultiSwitch Sprite (146) | 127 |
| The DoorTile# Sprite..... | 128 |
| The DipSwitch Sprite (162) | 128 |
| The ViewScreen Sprite (502) | 129 |
| The Crack1-Crack4 Sprites (546-549) | 129 |
| The Camera1 Sprite (621) | 129 |
| The CanWithSomething Sprite (1232) | 130 |
| The SeeNine Sprite (1247) | 130 |
| The Fem# Sprite (1312)..... | 131 |
| The Aplayer Sprite (1405) | 131 |

CHAPTER 7

Special Sector Effects 132

| | |
|--|-----|
| PREPARING TO DEFINE SPECIAL EFFECTS..... | 135 |
| Reviewing What You Know | 135 |
| Printing a Handy Reference | 136 |
| INTRODUCTION TO SECTOR EFFECTS | 137 |
| Creating a Simple DOOM-Style Door | 137 |
| Create Some Sectors | 137 |
| Assign a LoTag Value | 137 |

| | |
|--|-----|
| Try Out Your Door | 138 |
| Assign a Sound | 139 |
| Stop the Side Walls from Moving with the Door | 140 |
| Close the Door | 140 |
| Make the Door Close Automatically | 140 |
| Change the Door's Speed..... | 141 |
| Try Out Your Door Again | 141 |
| USING SECTOR TAGS..... | 143 |
| The Above Water (ST 1) and Underwater (ST 2) Sectors | 144 |
| Using an Underwater Sector (ST 2) Alone | 146 |
| Star Trek Doors (ST 9) | 146 |
| Elevator Transport (ST 15) | 149 |
| Elevator Platform Down (ST 16) | 153 |
| Elevator Platform Up (ST 17) | 154 |
| Elevator Car Down (ST 18) | 155 |
| Elevator Car Up (ST 19) | 155 |
| Ceiling Door (ST 20) | 155 |
| Floor Door (ST 21)..... | 155 |
| Creating a Floor-Based Door from Scratch..... | 157 |
| The Split Door (ST 22) | 158 |
| The Swinging Door (ST 23) | 158 |
| The Slide Door (ST 25)..... | 161 |
| Creating a Split Star Trek Door (ST 26) | 162 |
| The Bridge (ST 27)..... | 164 |
| Drop Floor (ST 28)..... | 164 |
| Teeth Door (ST 29) | 167 |
| Rotate Rise Door (ST 30) | 168 |
| Two-Way Train (ST 31) | 170 |
| Applying an Ambient Sound (ST 10000+) | 172 |
| Secret Areas (ST 32767)..... | 172 |
| Ending the Level (ST 65535)..... | 172 |
| USING SECTOREFFECTORS | 173 |
| Rotating Sector (SE 0) | 173 |
| Pivot Point for a Rotating Sector (SE 1) | 175 |
| Earthquake (SE 2) | 175 |
| Random Lights after Shot Out (SE 3)..... | 176 |
| Ceiling Lights Effect | 177 |
| Wall Lights Effect | 179 |
| Random Lights (SE 4)..... | 181 |
| Subway Engine (SE 6)..... | 182 |
| Teleport or Water (SE 7) | 183 |
| Sector Lights When Door Opens (SE 8) | 186 |
| Sector Lights When Door Closes (SE 9) | 187 |
| Door Auto-Close (SE 10)..... | 188 |
| Swinging Door Pivot Point (SE 11) | 188 |
| Light Switch (SE 12)..... | 188 |
| C-9 Explosive (SE 13)..... | 188 |
| Another Trick for Making Holes in Walls | 191 |
| Subway Car (SE 14) | 191 |
| The Slide Door (SE 15) | 191 |
| The Elevator Transport (SE 17) | 192 |
| Shot TouchPlate Ceiling Down (SE 19) | 192 |
| The Bridge (SE 20)..... | 193 |
| The Drop Floor (SE 21)..... | 193 |
| The Teeth Door (SE 22) | 193 |



| | |
|--|-----|
| Conveyor Belt or Water Current (SE 24) | 193 |
| Engine Piston (SE 25) | 194 |
| Camera for Demo Recording (SE 27) | 195 |
| Floating Sector or Waves (SE 29) | 196 |
| Two-Way Train (SE 30) | 197 |
| Floor Move (SE 31) | 197 |
| Ceiling Move (SE 32) | 199 |
| Earthquake Jibs (SE 33) | 201 |
| Shrink Ray Shooter (SE 36) | 201 |

CHAPTER 8

Special Construct Effects 204

| | |
|-------------------------------------|-----|
| WALL CONSTRUCTS | 206 |
| Creating Masked Walls | 206 |
| Glass Walls | 207 |
| Force Fields | 209 |
| Mirrors | 210 |
| SPRITE CONSTRUCTS | 211 |
| Vents | 212 |
| Bridges | 214 |
| Security Cameras and Monitors | 215 |
| Keys and Locked Doors | 216 |
| MISCELLANEOUS CONSTRUCTS | 217 |
| Space | 217 |
| Low Perimeter Walls | 218 |

CHAPTER 9

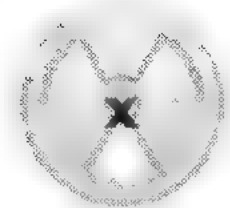
Designing Good Levels 222

| | |
|--|-----|
| DIFFERENT LEVELS FOR DIFFERENT GAMES | 224 |
| The Single-Player Design | 225 |
| Designing on Paper | 225 |
| Doing Initial Layout | 227 |
| Placing Major Items | 227 |
| Completing Structural Aspects | 228 |
| Placing Monsters | 229 |
| Placing Weapons and Ammo | 230 |
| Placing Power-Ups | 231 |
| Creating Scenery | 231 |
| Doing Final Testing | 232 |
| Good Level Design Considerations | 232 |
| Adding Suspense | 232 |
| Surprise, Surprise, Surprise! | 233 |
| Establishing Difficulty Levels | 233 |
| Building in Risk and Reward | 234 |
| Offering Puzzles and Solutions | 234 |

CHAPTER 10

Designing Co-Op and DukeMatch Levels 236

| | |
|---|-----|
| DESIGNING LEVELS FOR COOPERATIVE MODE | 238 |
| Design Open Spaces | 238 |
| Include More Monsters | 239 |
| Add More of Everything! | 239 |



THE DUKE NUKEM 3D Level Design Handbook

| | |
|---|-----|
| Design for Friendly Fire Effects | 239 |
| Include Explosives That Make You Go BOOM! | 239 |
| Develop Cooperative Puzzles | 240 |
| Set Up Divide and Conquer Situations | 240 |
| Use the Shrink Ray..... | 240 |
| Design for Unforeseen Situations | 241 |
| CREATING MEMORABLE DUKEMATCH LEVELS..... | 241 |
| Balance, Balance, Balance | 242 |
| Shrink the Level..... | 242 |
| Create a Nonlinear Design | 243 |
| Create Large Common Areas with Many Entry and Exit Points | 243 |
| Cut Down on the Puzzles | 243 |
| Ditch the Monsters..... | 244 |
| Establish Ambush Points | 244 |
| Use the Z Plane..... | 245 |
| Use Move Sectors | 245 |
| Add Sound and Other Cues | 246 |
| Give Players the Tools They Need to Set Traps | 246 |
| Set Wall-Mounted Laser Trip Bombs | 246 |
| Use C-9 Canisters | 247 |
| Give 'em Pipe Bombs | 247 |
| More Toys for DukeMatch | 248 |
| Shooters | 248 |
| Engine Pistons or Crushers | 248 |
| Jetpacks | 248 |
| The Holoduke | 248 |
| Cameras and Monitors | 249 |
| Water Sources | 249 |
| Night Vision Goggles (NVGs) | 249 |
| Example DukeMatch Levels | 249 |

CHAPTER 11

Using EditArt 250

| | |
|---|-----|
| THE UTILITY OF EDITART | 252 |
| Setting Up EditArt | 253 |
| Preparing the Graphic for Importation | 253 |
| Importing the Graphic into EditArt | 256 |
| Loading the Imported Graphic in Build | 257 |
| Animating Objects | 258 |

CHAPTER 12

Editing CON Files 264

| | |
|---|-----|
| DEFINING NUMBERS IN DEFS.CON..... | 266 |
| CHANGING DEFINITIONS IN USER.CON | 267 |
| WORKING WITH GAME.CON | 269 |
| The <i>actor</i> Keyword | 269 |
| Modifying Code | 272 |
| The <i>action</i> Keyword | 272 |
| The <i>move</i> Keyword | 275 |
| The <i>ai</i> keyword | 275 |
| The <i>state</i> keyword | 277 |
| CREATING NEW ACTORS | 280 |
| Adding Code to an Existing Sprite | 280 |
| Adding a New Enemy | 282 |

**CHAPTER 13****WAD2DUKE, A New DOOM WAD-to-Duke MAP Converter 286**

| | |
|---|-----|
| INTRODUCING WAD2DUKE | 288 |
| HOW WAD2DUKE TRANSLATES LEVEL DATA | 290 |
| A WAD2DUKE LEVEL CONVERSION DEMONSTRATION | 293 |
| Modifying Translation Results | 295 |
| Beyond the Yellow Door | 296 |
| The Small Computer Areas | 298 |
| The Water Vats | 299 |
| The Control Room | 300 |
| COMPLETING THE CONVERSION | 300 |

Appendix A: Making Memorable Levels 304

| | |
|---|-----|
| THE LEVELORD'S 17 DESIGN POINTS | 304 |
| 1. The Main Theme | 305 |
| 2. Frame Rate | 305 |
| 3. Continuity | 305 |
| 4. Details and Realism | 306 |
| 5. Appropriate Enemies and Weapons | 306 |
| 6. The Rule of the Bruces | 306 |
| 7. The Stick and the Carrot | 306 |
| 8. Pick Your Audience and Build around It | 306 |
| 9. Player Interaction | 307 |
| 10. A Well-Balanced Level | 307 |
| 11. Push the Envelope | 308 |
| 12. Patterns | 308 |
| 13. Connections | 308 |
| 14. Puzzles | 308 |
| 15. Think Sneaky | 309 |
| 16. The Fiddler's Roof | 309 |
| 17. Practice What You Preach | 309 |
| BLUM'S DUKE MAP CHECKLIST | 309 |
| Frame Rate | 310 |
| Single-Player Levels | 310 |
| Multiplayer Levels | 311 |
| Sounds | 311 |
| Secrets | 312 |
| Skill Levels | 312 |

Appendix B: Sounds List 313

| | |
|---|-----|
| COMPLETE LIST OF <i>DUKE NUKEM 3D</i> SOUNDS..... | 313 |
|---|-----|

Appendix C: CON File Statement Summary 330

| | |
|--------------------------------------|-----|
| COMPLETE LIST OF <i>DUKE C</i> | 330 |
|--------------------------------------|-----|

INTRODUCTION

Duke Nukem 3D, from 3D Realms Entertainment, is the latest first-person 3D shoot-'em-up computer game. Like all games before it in this genre, it takes the game-playing experience to new, previously unrealized heights. In this game, Duke Nukem, the character being played by the player, is more like a real person. As he battles his way through alien aggressors, he speaks, acts surprised, and even makes efficient use of restroom facilities. Most important, though, he is one bad dude who doesn't take any guff from those alien maggots.

Duke Nukem 3D's levels are technically superior to levels in prior games, which means that you the player have more realistic and exciting areas to explore. In this game, action can take place completely underwater as well as over it. With the aid of a jet-pack, you can fly high above the level's terrain and engage in aerial combat. Catwalks exist over floors, allowing you to stand directly over your enemies and ambush them from above. Moving subways can bring you clear across a level in seconds.

Computer games like this one have become more advanced over the last several years, not only in their game play, but in their expandability. Games like *Duke Nukem 3D* don't get put away when you, the player, defeat the last Boss on the last level and finish the game. Instead, you are given the tools to create your own levels—the same tools that the designers at 3D Realms themselves used to create the levels that come with the game. This means that from the moment you open the box and install the game, you can start designing devious new places for others to explore. These levels are stored in a compact file that can be easily shared with friends, neighbors, or the entire Internet community. The creation of new levels for games in this genre has spawned an entire cottage industry of level designers.

This book shall become your exhaustive reference on the use of these tools. You will learn how to make *Duke Nukem 3D* levels that take advantage of all the special tricks and design constructs available to you. You will learn how to create new artwork to import into the game, and you will learn how to reprogram the actions of all the creatures and objects. In fact, you can even create all-new creatures and objects, which allows you to customize the game in an infinite number of ways.

But that's not all this book will provide. My goal is not merely to teach you how to create new *Duke Nukem 3D* levels, but how to create *memorable* levels—levels that people will play over and over again, whether they're single players or several Duke-Matchers looking for a new level to serve as their battleground. To that end, you will

also hear from both of the level designers of all the original *Duke Nukem 3D* maps, Richard Bailey Gray (aka the Levelord) and Allen H. Blum III, who will impart their expert advice on what *they* think makes a good *Duke Nukem 3D* level.

Like all good game-related books, this one includes a CD-ROM packed with goodies to keep the avid Duke Nukem enthusiast entertained for hours on end. First and foremost, you get WAD2DUKE, a working converter program for converting all of those thousands of DOOM, Heretic, and Hexen levels out there into Duke Nukem 3D levels. The program is 100% configurable, too- so if you want to convert every medkit into a miniboss, fine! Next, you get two great CON file programming examples. The first shows you how to make the Pig Cops teleport in and out of danger, and the second shows you how to create code for an all new monster. For the programmers out there, there's some example code written in Delphi that shows you how to load and display a Duke Nukem 3D level. If your more artistic than analytical, then you'll find some artwork samples ready to import into Duke Nukem 3D. There's also 2 of the best Duke Nukem 3D "Frequently Asked Questions" files (FAQs) available anywhere. Finally, of course, there's a nice healthy dose of user levels, gathered from the four corners of the online world.

So, read on and get started with those really cool ideas for levels you've been tossing around. After all, as Duke Nukem would say,

"What are you waiting for, Christmas?"



The 3D First-Person
Shoot-'Em-Up Game Genre



The world of computer games keeps expanding and expanding at an almost incomprehensible pace. Each new game seems to outperform those released only a few months prior. The graphics become more spectacular, the sound is more awesome, and the features get more diverse. This evolving progression in technology is most evident in the 3D first-person perspective games, which generally involve annihilating enemies while traversing an ominous course from point A to point B.

BEFORE *DUKE NUKEM* WENT 3D

One of the earliest first-person games for the computer was *Wizardry*, by Sir-Tech Software. *Wizardry* featured a wire frame maze that a party of characters could explore. *Wizardry* featured very little, if any, animation. The maze was constructed on a grid, much like a sheet of graph paper. Each square in the grid was either solid, representing a thick wall, or clear, representing an area a player could walk through. From any square on the grid, one of four static views was generated, depending on if the players were facing north, south, east, or west. Monsters were represented as either single-frame or simple two-frame animation. Fighting was done by pressing the F key. You would get to strike the monster, then the monster would get to strike you. Then it was your turn again and then the monster's and so on until someone was dead.

Later, along came *Wolfenstein 3D* from id software. This game offered many new features in both graphics quality and game play that made it far superior to *Wizardry*. First and foremost, *Wolfenstein 3D* was one of the earliest first-person games to use a technology known as *texture mapping*. Texture mapping is a process whereby some



type of image is drawn onto a flat surface in a three-dimensional environment. The image, or *texture*, is said to be *mapped* onto the surface, hence the name. Texture mapping provided a new level of realism to computer games because it allowed a totally flat surface to appear like wood, rock, or water, simply by changing the image that was mapped onto that flat surface.

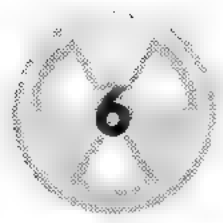
In addition to texture mapping, *Wolfenstein 3D* was a fully animated game. There was not a finite number of views available to be seen by the player. Instead, the game *rendered* a given view based on a player's location and direction, and whatever objects were in front of the player. This meant that objects, especially monsters, could move around on the map along with the player. The player had freedom to move in any direction along a plane. In fact, *Wolfenstein 3D* modeled movement so accurately for a game of its time that some people reported getting motion sickness by playing it.

Despite all of the advancements made by *Wolfenstein 3D* over its predecessors, the game still had some fairly large limitations. Walls were still restricted to only one length and had to be at 90-degree angles to one another. Floors and ceilings were all one height, which severely limited the types of rooms that could be created. All rooms were also the same light level, which left the levels feeling too uniform and stale.

THE *DOOM* LEGACY

The next game to really up the technology ante, of course, was id software's *DOOM* and games built on the *DOOM* engine, like *Heretic* and *Hexen*. *DOOM*'s basic premise was the same as *Wolfenstein 3D*'s: The player explored various levels in a 3D first-person perspective. However, *DOOM*'s graphics engine was rewritten from the ground up, instead of trying to expand the existing *Wolfenstein* engine. Although it was still at its core a texture-mapping engine like its predecessor, the new *DOOM* engine was a major advancement in graphics technology, providing much more varied and realistic levels.

The *DOOM* engine made it possible for the walls to be at arbitrary angles to one another, which allowed for nonrectangular rooms. The floors and ceilings of each room could also be at different elevations, which allowed for balconies, steps, pits, and other similar features. *DOOM* featured a much better lighting model, which allowed some rooms to appear darker and others brighter; this feature also allowed for more realistic shadow effects. Finally, the *DOOM* engine allowed floors and sectors to move up and down, which made additional features like elevators and crushing ceilings possible. This enhanced 3D environment served to immerse the player into the game more than any other game before it. People who got dizzy playing *Wolfenstein 3D* had to fight off waves of nausea playing *DOOM*.



Apart from the graphics engine, three other factors made *DOOM* the most popular game of its time. The first was its multiplayer ability. Gone were the days when the only opponents were computer-based. *DOOM* provided a human opponent to combat within its 3D world. Alternately, several players could play the same game as teammates, tackling the battles and puzzles within the game together. All that was required for multiple players were additional computers and a network or modem to send the game data back and forth.

The second factor contributing to *DOOM*'s popularity was id software's provision for expandability. If you had the means to create a new *DOOM* level, it was very easy to play that level in the game. Furthermore, id took a lenient stance toward programmers who wanted to write tools for creating levels. In fact, id software even published various parts of the *DOOM* level format so programmers could more easily write tools for users to create their own worlds.

The third event contributing to *DOOM*'s popularity was the dynamic growth of the online world, especially the Internet and the World Wide Web. People from all over the world could now share common interests via online services such as newsgroups, mailing lists, and FTP sites. This allowed level creators to share ideas, allowed programmers to share code so better utilities could be written, and allowed users all over the planet to upload and share literally thousands of new *DOOM* levels.

Even with all of the fantastic improvements that the *DOOM* graphics engine had to offer, there were still a few restrictions on the types of structures that you could create. Ceilings and floors had to all be parallel to each other, which meant no sloping floors or ceilings were possible. Also, for every x and y coordinate on a *DOOM* map, there could be only *one* floor z coordinate. This meant that it was impossible to create any structure directly over any other, which ruled out structures like catwalks or two-story buildings.

THE DAWNING OF *DUKE NUKEM 3D*

Well, this brings us to the present technology, and the ante has been upped again with the release of *Duke Nukem 3D* by 3D Realms Entertainment, a division of Apogee Software, Ltd. *Duke Nukem 3D* is the latest incarnation of the 3D first-person shoot-'em-ups. Its graphics engine, the great Build engine, allows for features the *DOOM*



engine couldn't perform. Build allows levels that have floors directly over other floors, and it can also provide sloped floors and ceilings. Sectors can move around, allowing for subway cars, swinging doors, and spinning gears. The lighting model is also more realistic, making the game even more intriguing. Those people who became dizzy from playing *Wolfenstein 3D* and *DOOM* can't even be contacted anymore; they're all locked in a small room somewhere taking Dramamine.

Let's look in a bit more detail at some of the features of this latest masterpiece in the 3D shoot-'em-up genre of computer games. Some of the features are similar to those that made the game's forerunners so popular, while some are new and unique to *Duke Nukem 3D*, offering an all-new game experience.

MULTIPLAYER ABILITY

Like many modern-day games, *Duke Nukem 3D* allows multiple players to play the game simultaneously. Although the game's packaging states that up to eight players can play the game at the same time, according to lead programmer Todd Replogle the eight-player limit is a practical boundary and *not* an absolute one. Replogle instead claims that as many as twelve players should be able to fire up a multiplayer game at the same time!

There are two multiplayer modes. The first is cooperative play, or Co-Op mode for short. In this mode of play, all the players form a single team and all are directed toward the same goal: eliminating the evil presence and finding the exit. The going is not always easy, however, because each of the players is capable of injuring, and being injured by, the other players. Friendly fire is a definite additional obstacle in these levels.

The second multiplayer mode is known as *DukeMatch* mode. In this mode, it's every Duke for himself. The goal is to eliminate all the other players and be the last one standing. As with many other video games, however, dying is not the final act that it is in real life, so a killed player is quickly reincarnated to start to battle again. The result is a fast-action slugfest in the most advanced 3D game engine to date!

WEAPONS

Duke Nukem 3D gives the player more weapons to choose from than any game before it. In all there are a staggering ten different weapons with which a player can dispatch

enemies. A player who has managed to collect each weapon is a walking armory indeed. The following list briefly describes each weapon:

❖ **The mighty foot** – This is Duke's own heavily booted right foot, and it allows you to *literally* kick some serious butt!

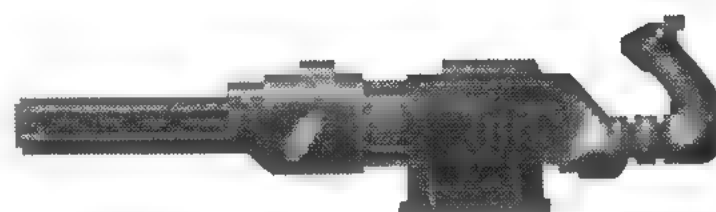
❖ **Handgun** – Each player starts off with a handgun, or pistol (along with the mighty foot, of course). This weapon can fire rapidly several times before Duke has to put in a new clip. The game handles this ammunition change automatically, but there is a slight pause as Duke loads the new clip. This pause often allows the current target to catch its breath and either retaliate or escape.



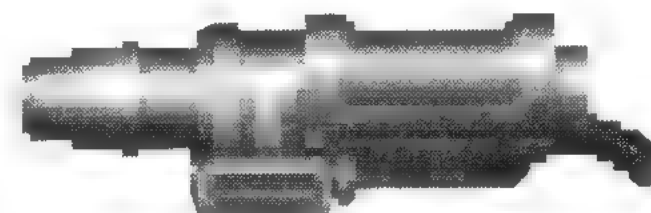
❖ **Shotgun** – This weapon packs a much more powerful punch than the pistol, but it also takes more time to reload. The shotgun is often used as the *basic weapon* for many players, or the weapon the player carries around most of the time.



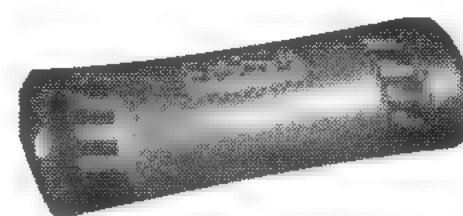
❖ **Ripper chaingun** – This baby fires about a dozen rounds per second. It's great for dispatching enemies, but it runs out of ammo quickly.



❖ **Rocket propelled grenade (RPG) launcher** – This powerful weapon causes a huge explosion upon impact, taking out most anything in its path with a single shot. The RPG launcher must be used with caution because it can also annihilate or injure critically the player who fires it, especially if it is fired at a close-range target or in tight quarters. For this reason, most discerning players use this as a long-range weapon only.

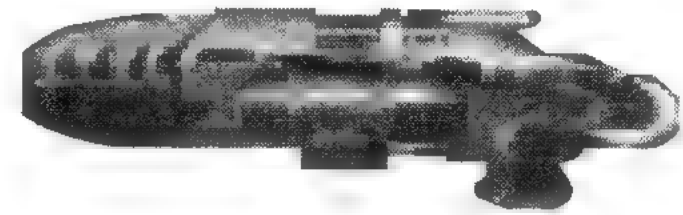


❖ **Pipe bomb** – This weapon is a true delight. Lay one of these down, wait for something to cross its path, and then detonate it from a safe distance.

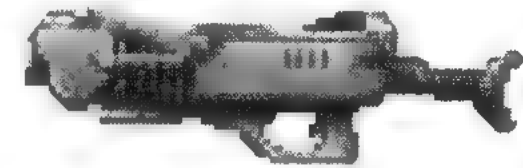


Small enough to remain unnoticed, this weapon is often extremely useful in DukeMatch mode.

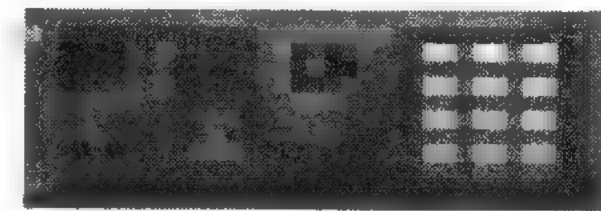
- ❖ **Shrinker** – This weapon instantly reduces most enemies to a few inches in height, after which they are easily compressed to a damp spot on the floor with a swift stomp of the mighty foot. The foot stomp is automatic once the player comes in range of a shrunken foe. If the enemy is not squished quickly enough, the foe will return to normal size in a few seconds.



- ❖ **Devastator** – This weapon fires several miniature missiles per second that can destroy almost every enemy in seconds. It is easily the most lethal weapon in Duke's arsenal.



- ❖ **Wall-mounted laser trip bomb** – This explosive device allows a player to lay a lethal trap. The device is placed against a wall, after which time it will emit a thin laser beam to an opposite wall. Any moving object that breaks the path of the laser sets off the trip bomb, causing a large explosive charge.



- ❖ **Freezethrower** – This weapon turns enemies into ice cubes. Once they are frozen, Duke will automatically kick his victim into a thousand fragments. He has to act fast, though, because the victim will defrost in a few seconds.



MONSTERS

The monsters in *Duke Nukem 3D* are as varied in number as they are deadly. Each creature has its own talents and its own nasty form of attack. You'll need to become familiar with each creature as both a *Duke Nukem 3D* player and as a game level designer. The following list briefly describes each enemy:

- ❖ **Assault trooper** – This is the basic alien patrol soldier who is often accompanied by several other troopers. This enemy can hover above you

if he's equipped with a jetpack. A trooper is fairly easy to kill, but he packs a pretty powerful laser pistol that can do its share of damage.

- ❖ **Assault captain** – This guy is very much like the trooper, but an assault captain also carries a device that allows him to teleport out of danger. He often teleports back into battle right behind his opponent.

- ❖ **Pig cop** – This brawny mutant wields a nasty shotgun, and it often takes several shots to bring him down. Pig cops can also be found in hovering recon patrol vehicles (RPVs).

- ❖ **Octabrain** – This bizarre creature can both fly and swim with equal dexterity. An octabrain's powerful mind blast can reduce you to a puddle of mental goo. It can also deliver a brutal bite at close range.

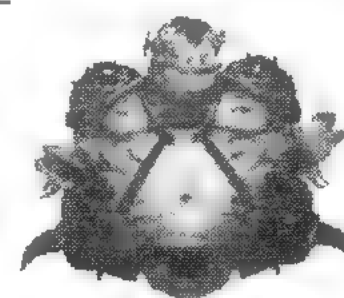
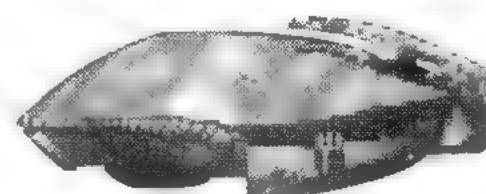
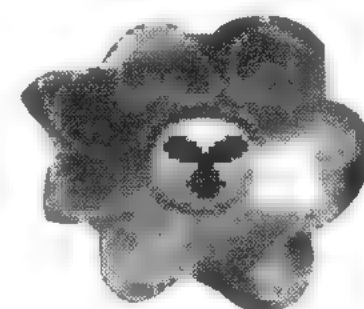
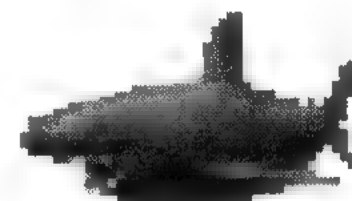
- ❖ **Shark** – A bit more ruthless than a real shark, this critter goes for blood at every opportunity.

- ❖ **Protozoid slimer** – This enemy is a gelatinous puddle that feeds on your innards. This egg-spawned nightmare quickly hops from ceiling to floor, making it extremely difficult to shoot. One shot with any weapon usually takes it down, if you ever do get a bead on one.

- ❖ **Sentry drone** – This is a robot-controlled pod programmed to explode kamikaze-style after hunting down its target. A sentry drone is also programmed to avoid incoming fire.

- ❖ **Enforcer** – This is a lizardlike alien humanoid that can either open fire with its ripper chainguns or spit a viscous glob of poison at your face. It can also leap high into the air, making it more difficult to track down.

- ❖ **Assault commander** – This guy is a rotund alien that hovers freely on antigravity pads. He packs a large rocket that he can aim with amazing precision.



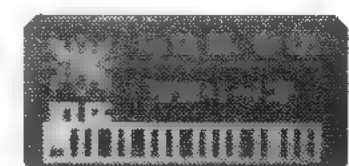
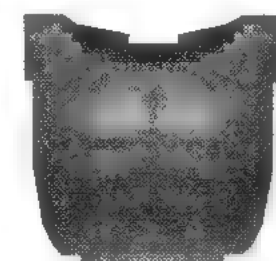
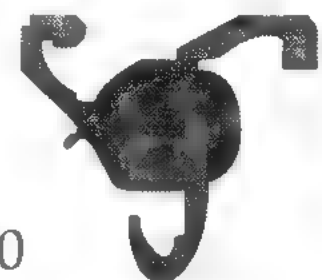
- ❖ **The Bosses** – There are three of these supreme alien beings to conquer. I won't give away all of their secret forms of attack for now, just in case you've yet to do battle with them. Suffice it to say, anyone trying to take on these guys face to face might have a screw loose.



POWER-UPS

The various power-ups available to resourceful players provide a surplus of splendid effects, which you will no doubt want to include in your own level designs. These items help complement strategy and prolong players' enjoyment by boosting health, providing access to otherwise restricted sectors, and aiding mobility in sectors that contain challenging terrain or environmental conditions. The following list provides a brief description of each power-up:

- ❖ **Medkit** – This power-up is available in small and large sizes, with smaller ones generally being more prevalent. Small medkits increase health by 10 or 30 percent instantly. Large medkits contain a full 100 percent of health from which players can replenish their health levels up to 100 percent when they activate it. The amount it takes to replenish a player's health to 100 percent is deducted from the large medkit until it is used up.
- ❖ **Atomic health unit** – Resembling glowing atoms, this lucky find increases a player's health to 50 percent, even if the increase takes the player's total health level above 100 percent. This power-up can maximize a player's health up to 200 percent!
- ❖ **Armor** – This power-up allows a player to absorb more punishment from enemy barrages before it begins to diminish the player's health level.
- ❖ **Keycard** – Available in red, yellow, or blue, this item allows a player to unlock a door or gate latch to provide

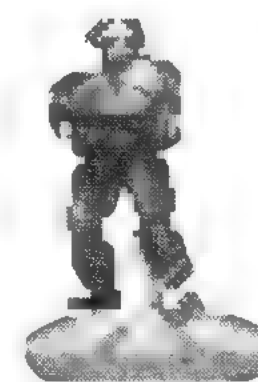


entry into otherwise closed sectors containing vital weapons, power-ups, or perhaps a better route to the level's much-sought-after exit.

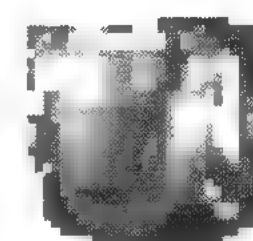
- ❖ **Steroids** – A player must activate this power-up (after finding and possessing it) to gain a brief period where energy, speed, and resiliency to enemy fire are boosted.



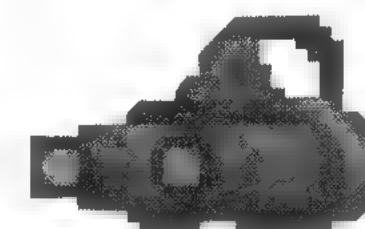
- ❖ **Holoduke** – This power-up is a holographic likeness of Duke himself that allows a player to strategically divert the enemy's attention away from the real Duke temporarily.



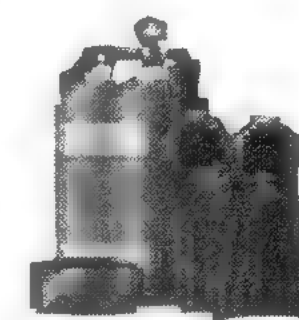
- ❖ **Jetpack** – This extremely useful power-up allows a player to become airborne for making quick getaways or commencing hovering attacks.



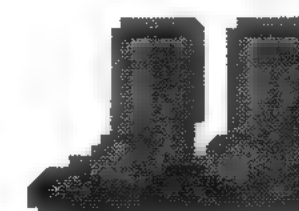
- ❖ **Night vision goggles (NVG)** – Because some sectors and even entire levels can be poorly lit, this power-up enables a player to more confidently traverse darker passages and identify lurking enemies.



- ❖ **Scuba gear** – The game's great variety of terrain includes plenty of water, and Duke can only hold his breath for so long before a player's health level is effected. This power-up activates automatically when a player submerges, supplying enough oxygen for extended underwater maneuvering.



- ❖ **Protective boots** – Toxic sludge canals and intensely hot lava rivers add to the variety of terrain obstacles a player must overcome. Like the scuba gear, this power-up provides automatic protection from these environments for extended time periods.



ADDING YOUR OWN TOUCH

If you've played *Duke Nukem 3D* to its full extent, your creative side has probably taken hold and you've decided to try and use the utilities that come with the game. Unfortunately, creating levels is not an easy task. A *Duke Nukem 3D* level is composed of many different individual data elements, and these elements interrelate in many different ways to create all the different types of structures and effects that you see during game play.

Perhaps you have wondered how the marvelous effects and architectural constructs were created. This book will hopefully illuminate these mysteries and grant you the title of supreme *Duke Nukem 3D* level creator. Your levels will inspire awe throughout the land. You will become rich, famous, and powerful. You will own a mansion and a yacht...wait, I'm getting ahead of myself, here. Before you can do any of those things, you'll need to read this book. To quote Duke one more time...

It's time to kick ass or chew bubblegum,
and I'm all out of gum. —*Duke Nukem*





Introduction to Build



As you already know, the retail version of *Duke Nukem 3D* comes with its own level editor. The editor is named after the game engine that *Duke Nukem 3D* is based upon, Build. As with any program ever written, Build has its good points and its not-quite-as-good points, so let's take a look at both its benefits and its limitations.

BUILD'S GOOD POINTS

Build includes among its benefits a 3D view mode for viewing the progress of level design in addition to the 2D view mode in which design takes place. Build is also easy to use after you become familiar with its exhaustive list of keystrokes and commands. The Build level editor provides a couple of example levels that help you to hone your skills as you become acquainted with the program. Let's now take a detailed look at each of these benefits.

VIEWING THE DESIGN IN 3D MODE

Unlike any *DOOM* level editor ever written, Build lets you walk through your level as you create it. This allows you to see almost exactly what each area is going to look like when a player will be playing your level in a game situation. Build's 3D view mode probably cuts in half the total time needed to design a given area because you don't have to quit Build and load the game to see your work in progress. Many *DOOM* level editors are good 2D editors, but the level designer is forced to save the level, exit the editor, and load *DOOM* to see exactly how the textures look on the walls and how light or dark the brightness is in each area. The time spent saving the map and going between programs adds up very quickly, creating a lot of wasted design time. To make matters

worse, a *DOOM* level requires a complicated structure called a *Binary Space Partition*, also known as a *BSP tree*, to be created so it can display the level properly. These BSP trees often take several minutes to generate, and they have to be regenerated every time a wall is moved. This further adds to the time necessary to build the level.

None of these problems exist when using Build, however, because of its 3D mode. Half of all your level development will be done in 3D mode, and you will become quite accustomed to walking through your own level *as you create it*, much like a construction supervisor overseeing the work in progress.

Figure 2.1 shows you the 3D preview of the *DOOM* level editor that I wrote, named DOOMCAD, and Figure 2.2 shows you the 3D view mode available in Build. I don't mind one bit saying that the latter view is superior for level editing to the view offered by my own program.

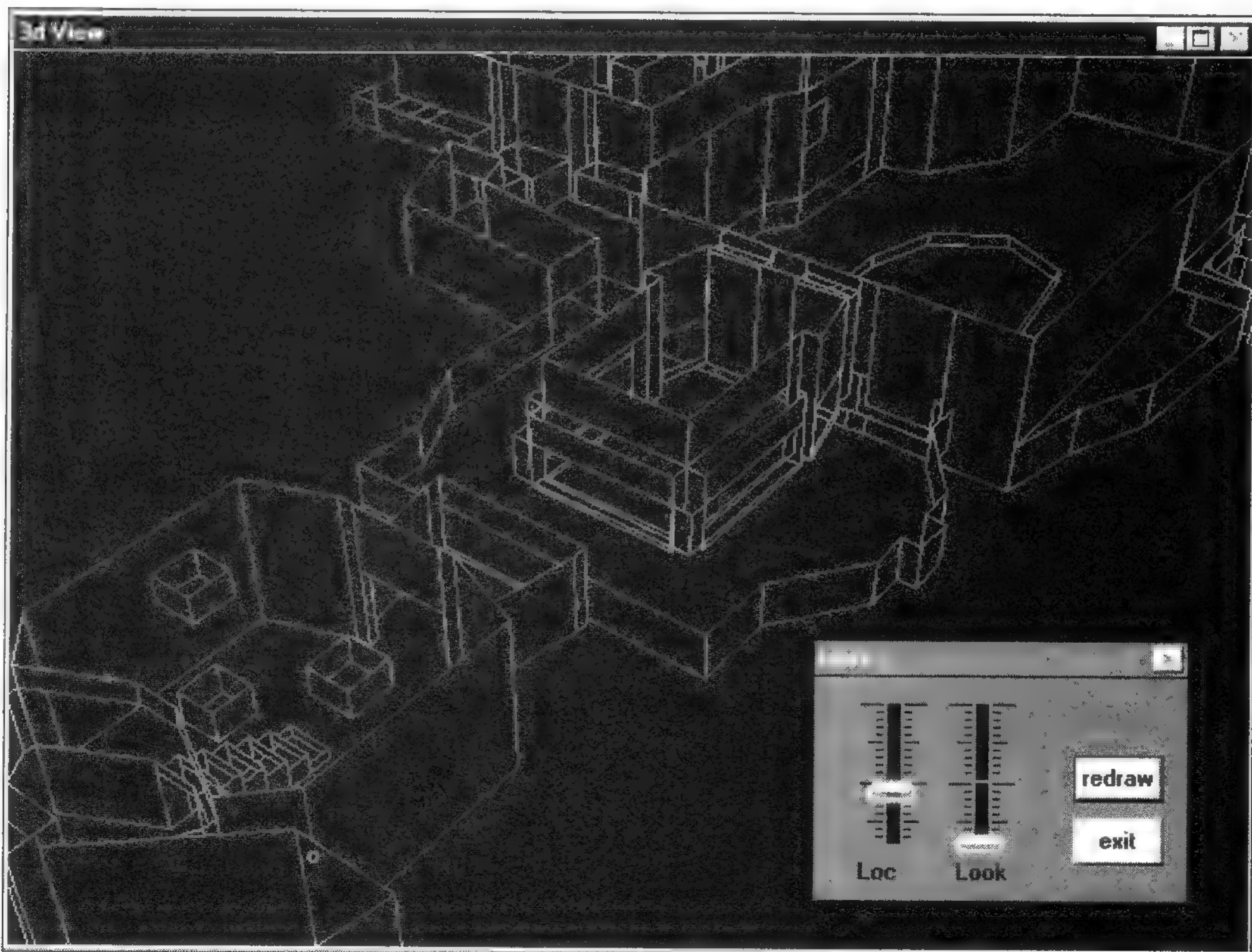


FIGURE 2.1: DoomCAD's 3D preview was functional but quite limited compared to Build's 3D mode.



FIGURE 2.2: Build's 3D mode appears to be a bit superior, don't you think?

DESIGNING OPERATIONS IS EASY

Once you know how to use Build, you will find it extremely easy to use. For example, moving walls around is accomplished easily with the mouse. If you have any experience with any mouse-driven program (does anyone *not* have experience using a mouse?), you will find Build very easy to use. The difficult part of using Build is the initial learning curve, which will be discussed shortly.

WORKING WITH BUILD-PROVIDED EXAMPLES

Build provides two example files, `_SE.MAP` and `_ST.MAP`, which demonstrate each and every one of the special effects that are possible in the game. You can load these levels into Build to study them or play them in the game itself. You will probably want to do both. When you look at each of these levels, you will be presented with a set of numbered teleporter pads. Each pad will take you to a room demonstrating one of the special functions the Build engine can perform. Because each of these functions is also numbered, you can match the number in your documentation with the number on

the teleporter pad and check out how to create that special function. This makes for an excellent learning tool.

In addition, a third map named `_ZOO.MAP` is supplied that demonstrates all of the objects that you can encounter during play. This is another good source for seeing all of the possible items with which you might populate the levels you design.

BUILD'S LIMITATIONS

Although Build is clearly a superior level editor, the program does have certain limitations. Each of its less-than-superior qualities should also be considered by anyone who is serious about crafting quality level designs.

WORKING WITH BUILD'S HOT KEY-BASED INTERFACE

As mentioned above, using Build is very easy once you're familiar with it; however, the caveat here is that becoming familiar with the program may take a bit longer than becoming familiar with some other programs. The reason for this is that almost every function you can perform in Build is done by using a specific key sequence, and there are over 100 different functions in Build. Memorizing the functions of over 100 different sequences of keystrokes can be a daunting task. Of course, many of the keystrokes represent the first letter of their function. Pressing the S key inserts a new sprite object, for example. Other functions are not so easily matched because the program's author soon ran out of letters for all the functions. Often, questions like, "Was it F, Alt + F, Ctrl + F, or Right Alt + F to flip a texture?" were typical of many that I asked myself as I tried to create my first Build structures. Eventually, you'll find that you memorize all of the myriad keystroke combinations. However, the sheer number of keystrokes to memorize makes the initial learning curve a pretty steep one.

WORKING WITH "NOT-SO-FRIENDLY" DOCUMENTATION

Although the documentation accompanying Build serves as an adequate reference for a somewhat experienced level author, a beginner may find all the different terminology and instructions intimidating. Even 3D Realms itself admits that the Build documentation, which can be found by running the program `BUILDHLP.EXE`, was written "at the last minute." Another problem in `BUILDHLP` is that a few typographical errors



that exist in some of the keystroke commands escaped proofreading, which resulted in some inaccuracies. Therefore, don't be surprised if the documentation tells you to press a particular key to perform a particular action, and when you do it nothing—or perhaps something unexpected—happens!

However, this limitation shouldn't hold you back. Beefing up and enhancing the Build documentation is precisely the mission I hope to accomplish with this book, so I won't belabor this point any further.

SEEING MOVING SECTORS IS NOT POSSIBLE

Although the 3D view mode in Build is great for previewing *most* of your level, any special moving sectors like doors and elevators will not function until you try them out in the game itself. This too is only a minor problem, however, because you're going to have to go into the game to actually play-test your level anyway, and you'll be able to try out all the moving sectors while you play-test.

STARTING BUILD

Now that you've learned a few things about Build, it's time to go into the program and check it out for yourself. The first thing you need to do is find Build on the *Duke Nukem 3D* CD-ROM. The Build editor is in the subdirectory (or folder) named *Build* that is in the directory (or folder) named *Goodies*. To run Build, you need to first copy all the files in its directory to the directory on your hard drive that contains the *Duke Nukem 3D* program files. Note that the Build files *must* reside in the same directory as the game; the level editor will *not* function properly if you attempt to run it from another directory. Assuming you have installed *Duke Nukem 3D* to its default location (the directory named Duke3d), the following DOS command will copy all the proper files into the correct directory.

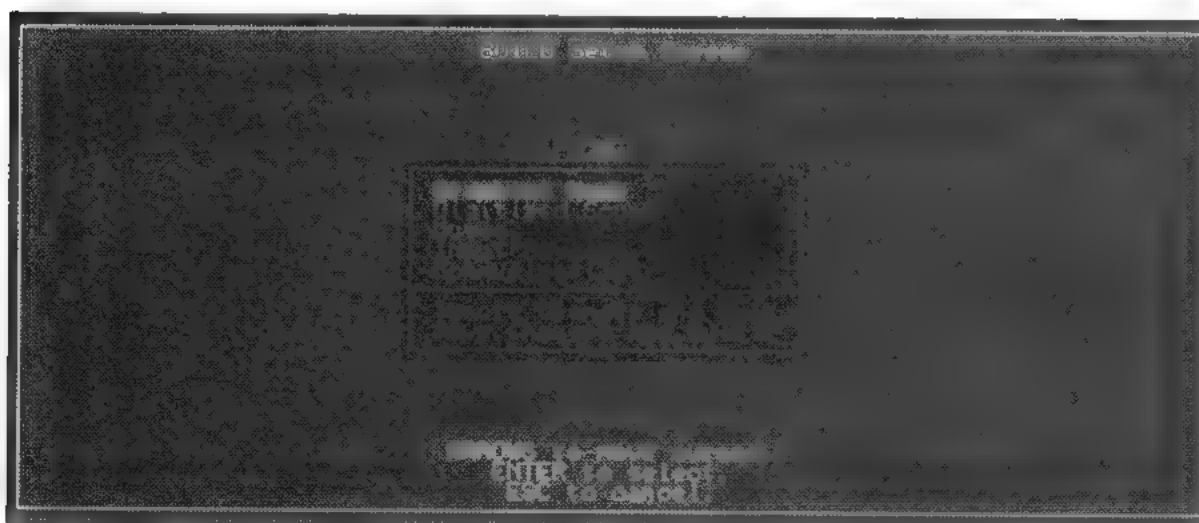
```
COPY D:\GOODIES\BUILD\*.* C:\DUKE3D
```

In this example, D: represents the drive letter of your CD-ROM drive. If your CD-ROM drive is identified by another letter, then you need to substitute the correct letter for the D: in the above example. Similarly, if your *Duke Nukem 3D* directory is not Duke3d, then replace that part of the command with the correct name of the directory in which your Duke Nukem program files reside.

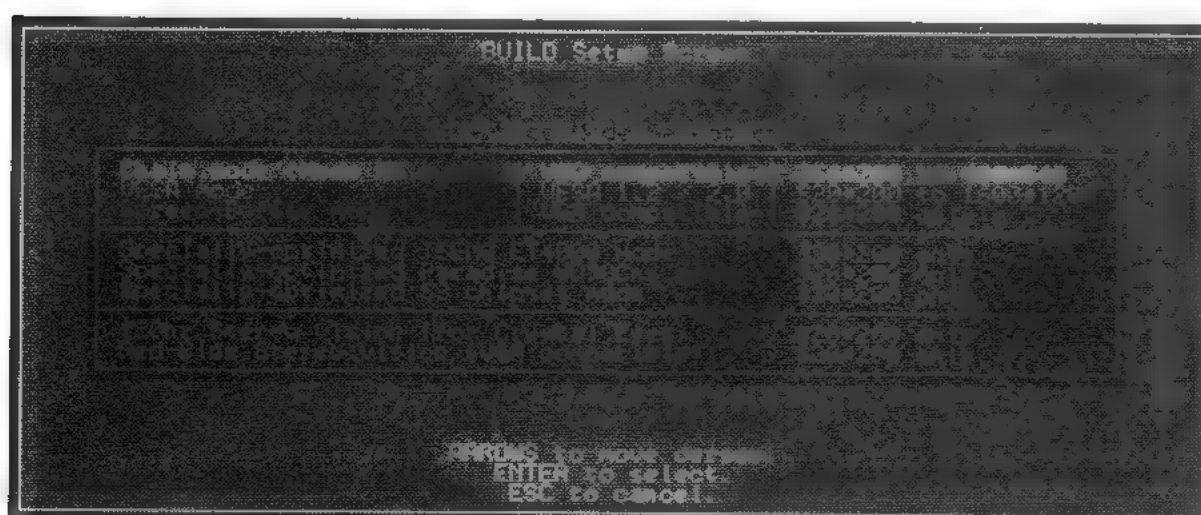
You'll need to do one more thing before you can start Build. When you copy files from the *Duke Nukem 3D* CD-ROM over to the hard drive, DOS marks these files as read-only. Therefore, you'll need to remove the read-only attribute from all of the Build files because you are going to have to write information to some of these files. The following DOS commands will accomplish this:

```
C:
CD\DUKE3D
ATTRIB -R *.*
```

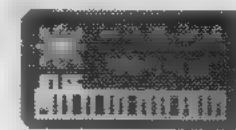
Next, you will want to set up the video mode that Build will use. If you can run *Duke Nukem 3D* in a higher resolution, you may also want to run Build in that same resolution. To set the video mode, run the BSETUP program from DOS. You will be presented with the following choices:



Strangely enough, the only setting that will have any effect is the first one, Graphics Mode. The sound, input, and communications menu options do absolutely nothing. Perhaps this menu was the original menu for *Duke Nukem 3D* itself, and the level editor was going to be built right into the game. Regardless of whatever reason these extra options exist, you can ignore them. When you select the Graphics Mode option, you will be presented with the following menu:



Choose a video mode your hardware can support. If you're at all unsure about your video hardware, select the first option, and then, on the following menu, make sure to select an X value of 320 and a Y value of 200. These are the default settings for a standard VGA card, and most computers built in the last eight years can support it.



NOTE

Again, if you installed the game's program files to your hard drive in a directory with another name, be sure you change to that directory before typing the ATTRIB command.

After you decide on a video mode, select “Save Changes and Quit” from the first menu, which will save your video setting and bring you back to the DOS prompt. You’re finally ready to check out what this baby can do!

Finally, to go into Build for the very first time, simply type **BUILD** at the DOS prompt and press Enter. You must be in the main *Duke Nukem 3D* directory when you run this command because this is the directory where you copied the Build files. If this is the very first time you’ve run Build, you’ll see the screen shown in Figure 2.3.

BUILDING YOUR FIRST LEVEL

The opening Build screen, shown in Figure 2.3, displays Build’s 2D view mode. What you should see is a large grid area, and a gray status bar at the bottom of the screen.

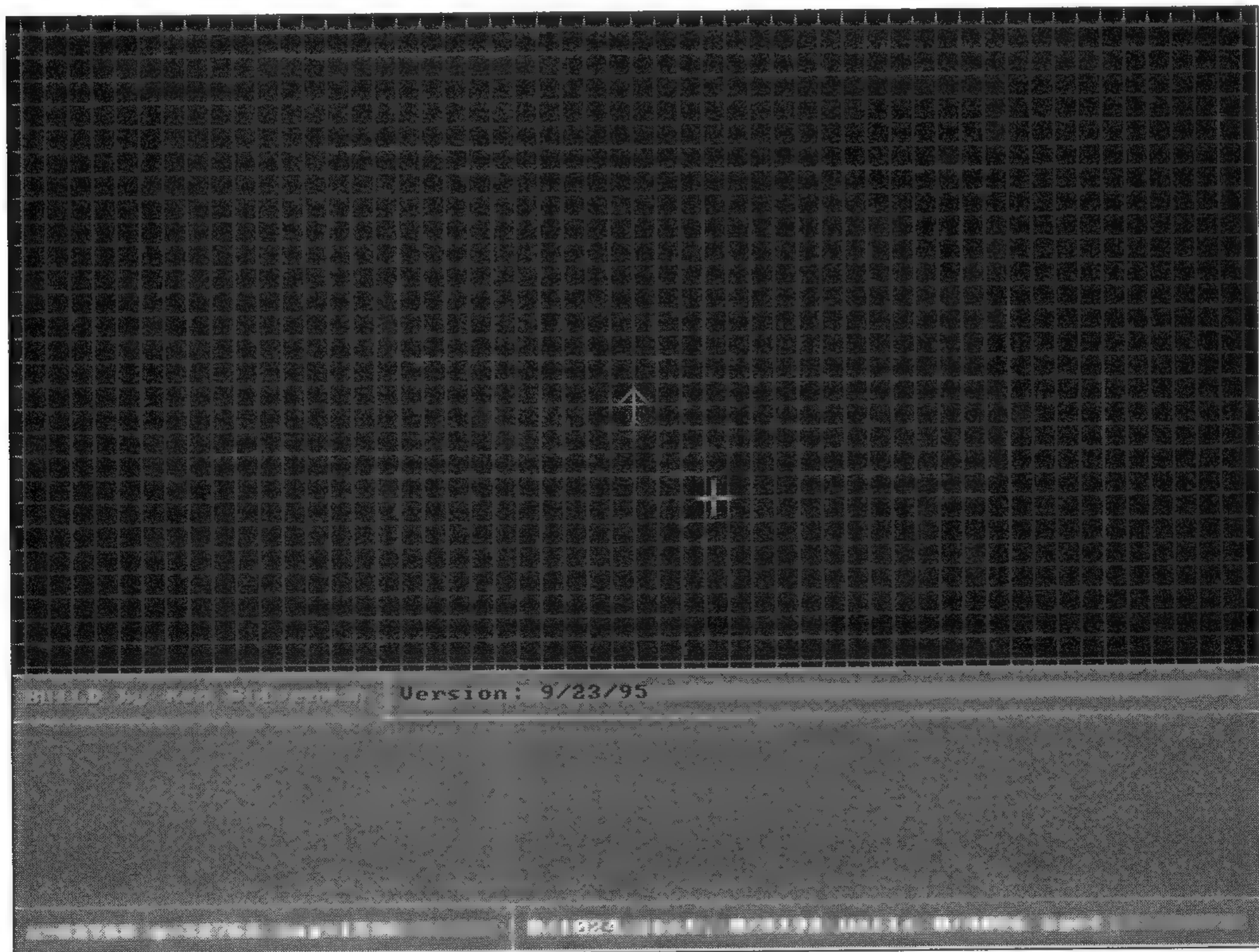


FIGURE 2.3: This is the opening Build screen in which you may begin designing a level.

Within the large grid area should be a pink crosshairs representing the mouse cursor, the grid, and a white arrow. For now do not be concerned with the white arrow; we'll get to it later. The grid is where you will begin designing your own level. Before you do, however, let's get a few definitions established so the instructions that follow will be clear.

THE ELEMENTS OF A SECTOR

Your first experience in designing a level will be creating a rectangular room. The room will be made of four walls, a floor, and a ceiling. The components that will make up the room are defined as follows:

- ❖ Each of the anchor points at the corners of the rectangle is called a *vertex* (multiple anchors are called *vertices*). You will soon draw four vertices to create a room.
- ❖ Each of the four white lines that makes up the room is called, obviously enough, a *wall*.
- ❖ Finally, the entire rectangle itself is what's known as a *sector*. The official definition of a sector is any enclosed area of space on the map that has a common floor and ceiling.

Sectors are the basis of all *Duke Nukem 3D* level maps. The rooms and structures that you create will be made up of one or more sectors. The room you are about to create will be made up of a single sector. However, it is possible that a single room or structure can be made up of many sectors. In fact, this is true most of the time.

CREATING A SECTOR

First, use your mouse to move the cursor somewhere above and to the left of the white arrow. After you get the cursor there, press the spacebar, and move the mouse again. You'll notice that when you press the spacebar, the message "Sector drawing started." appears on the status bar right under the view window. When you move the mouse now, notice that a white line follows the mouse cursor, with its starting point at the first vertex where you pressed the spacebar, as shown in Figure 2.4. The message "Sector drawing started." tells you that you are in what's known as *sector drawing mode*.

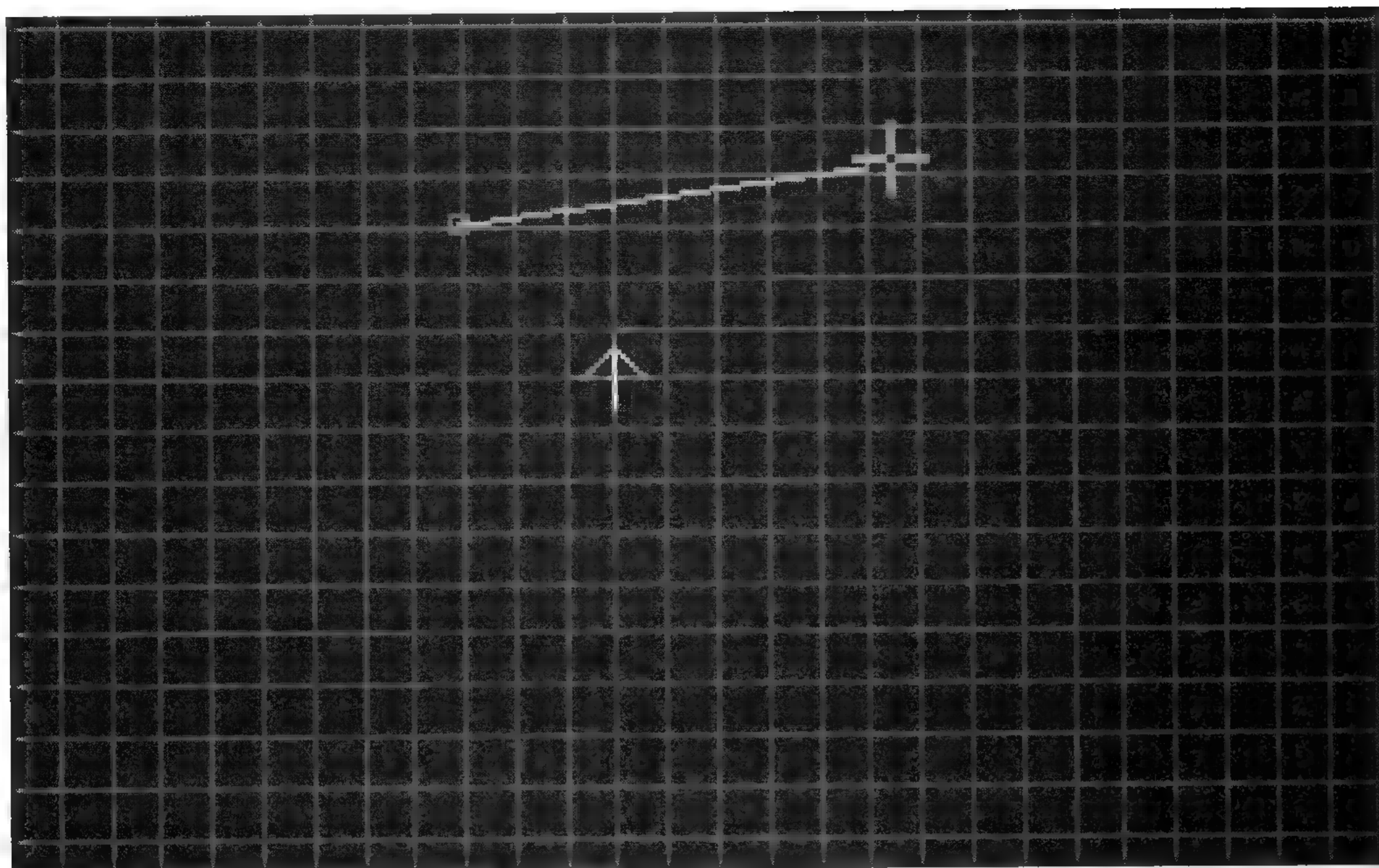
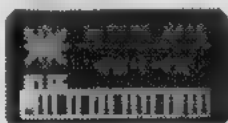


FIGURE 2.4: Position the mouse cursor, press the spacebar, and then move the mouse to establish the first, upper-left corner, vertex.



NOTE

If you make a mistake and press the spacebar on a place where you didn't want to, you can easily *unanchor* the vertex you just created by pressing the Backspace key. This is especially useful when you are trying to place the last vertex of the sector exactly on top of the first one. You can also press the Backspace key multiple times in succession, and each time it will remove the previous vertex, until no vertices are left. When you delete the last vertex in this manner, you will automatically leave sector drawing mode.

Move the mouse to the right and above the white arrow, but along the same horizontal grid line on which you anchored the first vertex, and press the spacebar again. Once again, a vertex will be anchored at the spot where you pressed the spacebar, and moving the mouse cursor will start drawing the next white line (wall), as shown in Figure 2.5.

Your eventual goal is to draw a rectangle around the white arrow. Once you get to each corner of the rectangle, press the spacebar to anchor a new vertex there, and start drawing the next line. Your last press of the spacebar should be on the same vertex as the first corner of the rectangle, in the upper-left corner. Once you press the spacebar on a vertex for the second time, sector drawing mode will stop automatically.

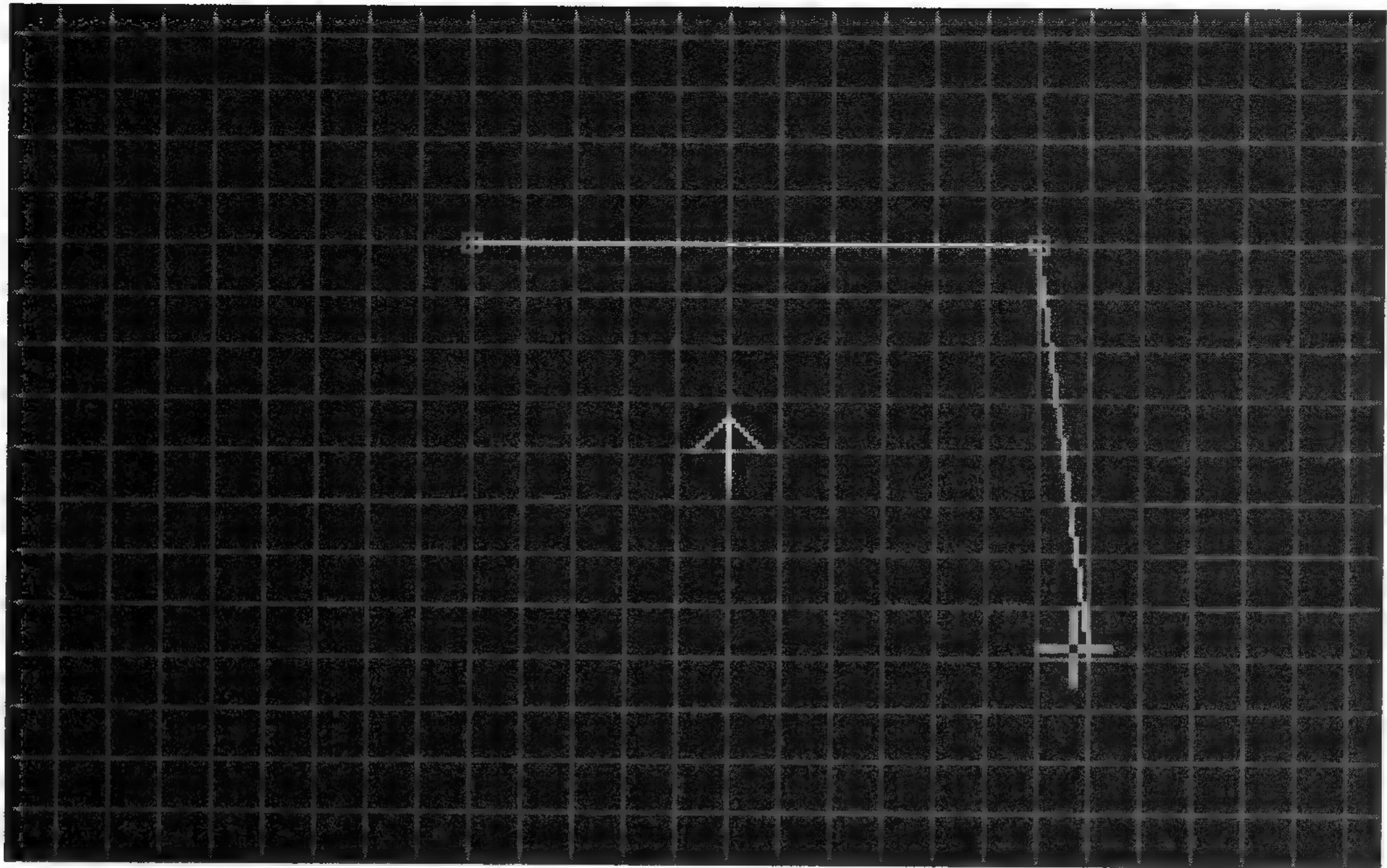


FIGURE 2.5: Anchoring the second vertex establishes the first (north) wall of your sector.

All in all, you should have pressed the spacebar five times: once each in the upper-left, upper-right, lower-right, and lower-left corners and then one more time back in the upper-left corner to end the operation. The final product should look similar to the one shown in Figure 2.6.

STUDYING ACTUAL GAME SECTORS

You've just drawn your very first sector! Now let's look at a few examples of some areas from original *Duke Nukem 3D* maps and look at all the sectors that make up those areas.

Sector Case Study #1: The Saloon

The room you see in Figure 2.7 is a 2D view of the saloon in The Red Light District level in L.A. Meltdown (E1L2), the game's first episode. (All the objects have been removed for clarity.) Figure 2.8 shows a shot of the same area taken in 3D view mode. Study all the enclosed spaces that make up the saloon. Look at the bar itself. Note that the bar is primarily made up of a U-shaped sector. When you look at the bar in 3D

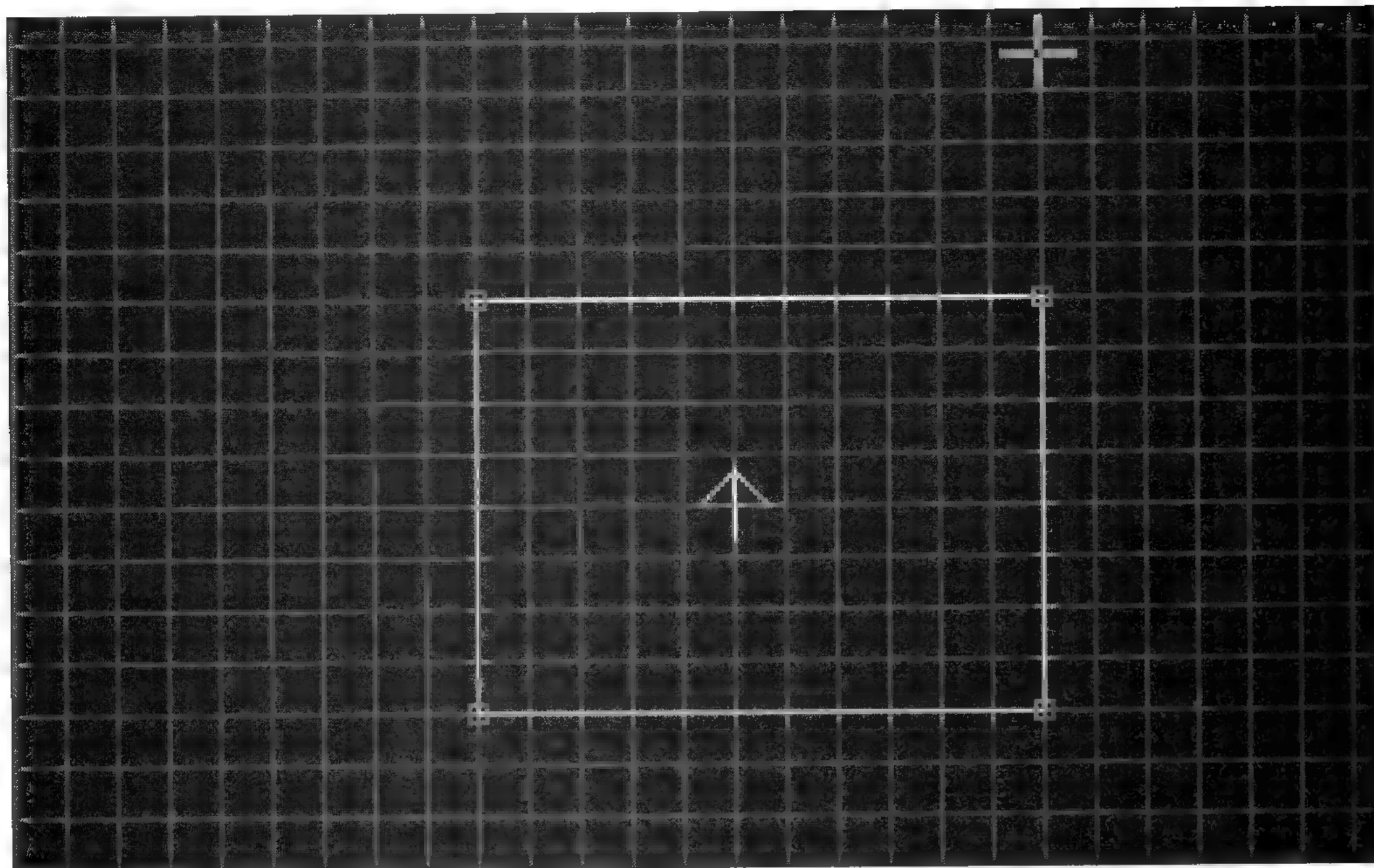


FIGURE 2.6: The rectangular sector is completed when you set all four vertices.

mode, you'll notice that it is an area of space that has a lower ceiling than the rest of the room as well as a higher floor (the top of the bar is actually the *floor* of the sector). Each of the architectural features in the saloon is made up of sectors that have varying floor and ceiling heights. Let's go through each of the features now.

- ❖ **The monitor** (purple line at the saloon's west wall) – Several monitors appear on each level to give the player glimpses into other areas, courtesy of surveillance cameras. Here there is a small sector that the view monitor resides in, off of the west wall of the saloon. The floor of this sector is lower than the room, and the ceiling is higher, making a small recess in the west saloon wall.
- ❖ **The television** (northeast corner) – The TV with the infamous white Bronco is also its own sector. In this case, the floor of the sector matches the floor of the room, but the ceiling is lowered. The exposed surface created by lowering the sector's ceiling becomes the four sides of the TV.

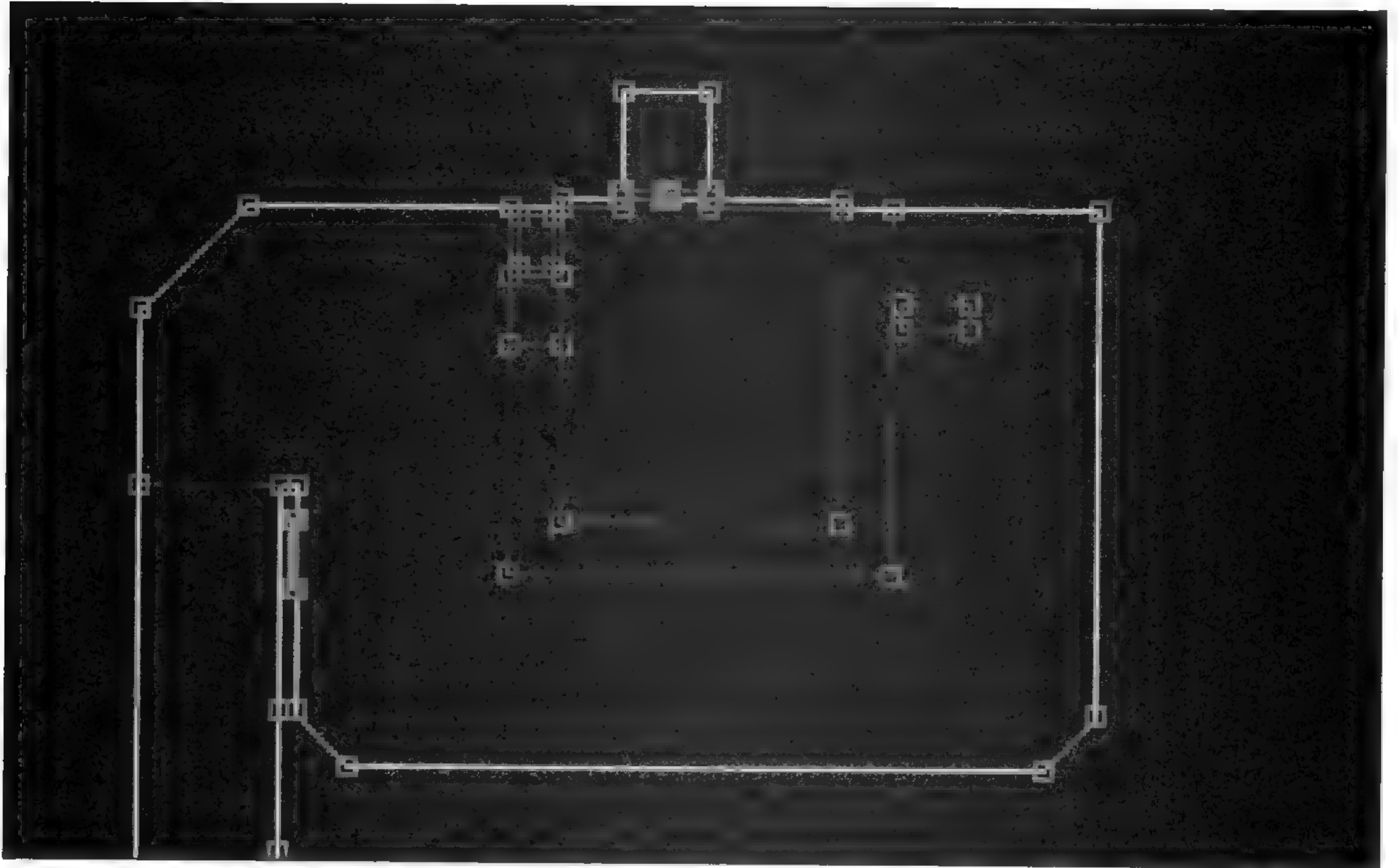


FIGURE 2.7: The Red Light District's saloon area is composed of several sectors.

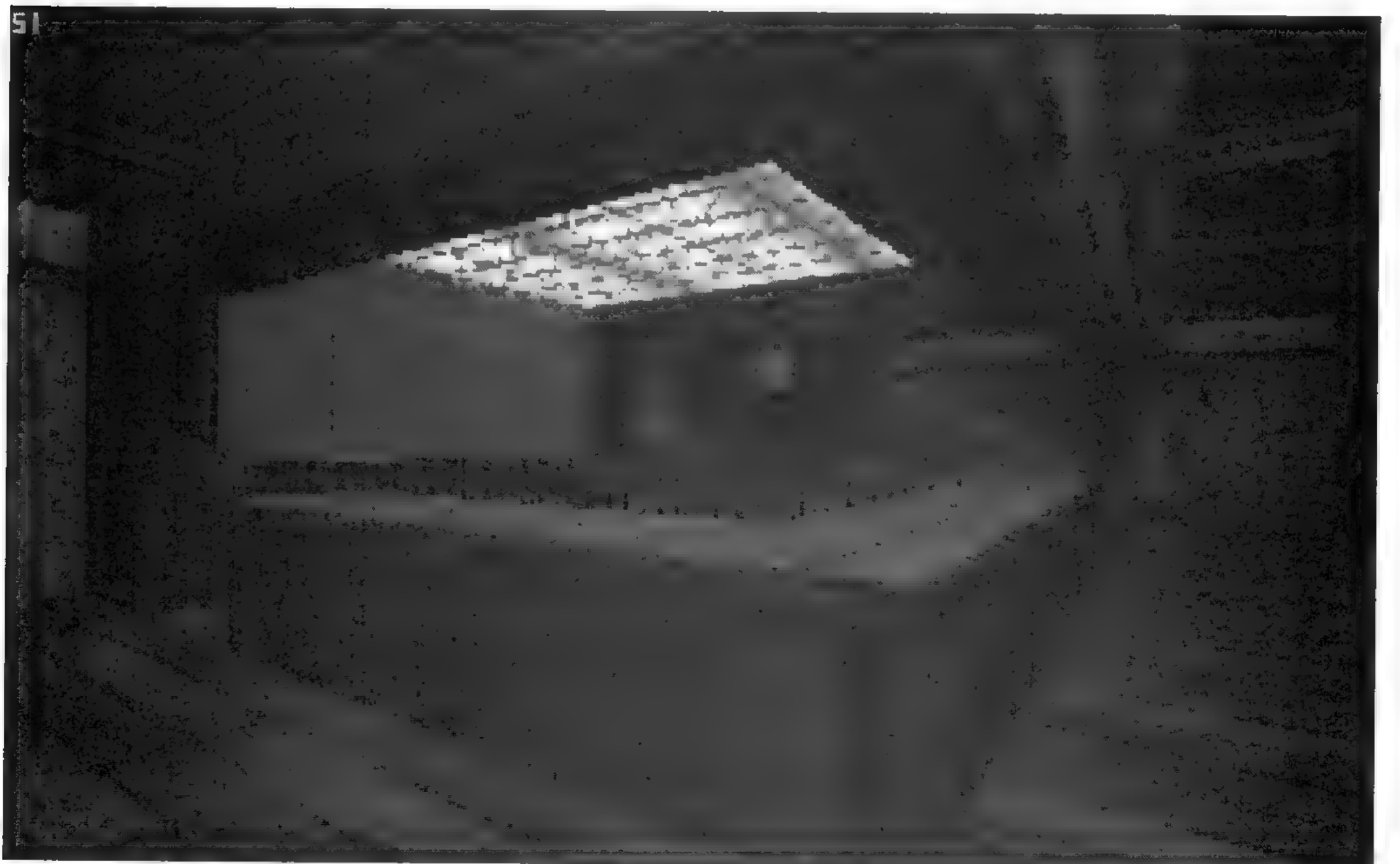


FIGURE 2.8: This is how the saloon area looks in 3D from the southwest corner of the room.



- ❖ **The cash register** (northwest corner of the bar) – This sector lies inside the bar sector. Its ceiling is the same as the bar sector, but its floor is higher. In addition, the floor of the register is *sloped*, or angled in a certain direction. This angled surface makes up the front part of the register where the register keys would be.
- ❖ **Hidden compartment** (against the north wall, inside the bar area) – This little compartment houses the red keycard. It is made by lowering the ceiling so that it is much lower than the saloon's ceiling and keeping the floor the same height.
- ❖ **The exit** (far west side) – The exit is another sector with a sloped floor, creating a ramp.

By studying Figures 2.7 and 2.8, you should be able to tell the location the player was standing in from the perspective shown in the 3D snapshot. Try to locate that spot on the 2D portion of the map. You should be able to tell that the player was standing in the southwest corner of the room.

Sector Case Study #2: The Submarine

In the saloon, you saw several types of architectural features that can be created just by making sectors of different shapes and sizes. These features can be very simple, like the television described above, or they can be extremely complicated, like the submarine in the Death Row level in L.A. Meltdown (E1L3) (see Figure 2.9). The submarine is made up of 17 different sectors of various shapes and sizes, as shown in Figure 2.10. In each of these sectors, the ceiling is the same height and has an outdoor texture, which suggests that this area is outside, under a starry sky. The floor of each sector varies in height and in slope, however, which gives the different parts of the submarine its shape.

Sector Case Study #3: The Pagan Chapel

Another good room in which to study the different ways that sectors can be used is the pagan chapel, which is also on the Death Row level. The pagan chapel is a single room, yet it is composed of several dozen sectors, as you can see in Figure 2.11. The ceilings of most of these sectors are sloped to form what appears to be a peaked roof.



FIGURE 2.9: This is Death Row's submarine area in 3D view mode.

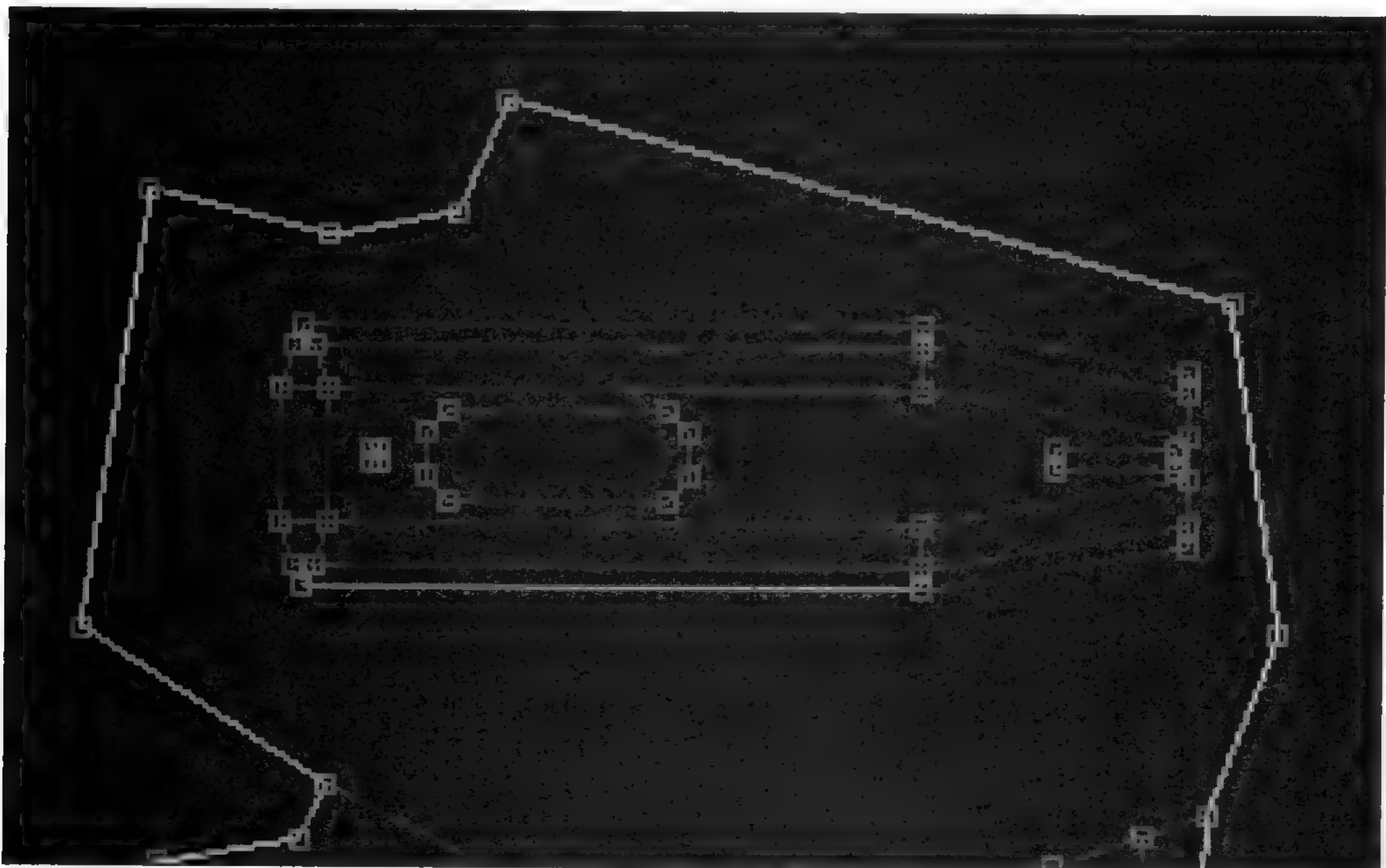


FIGURE 2.10: This 2D view of the submarine area shows it is made up of 17 sectors.

**TIP**

You can load any of the levels from the original game by typing **BUILD ExLy** at the DOS command line, where **x** is the episode number (1-3) and **y** is the level number (1-11). The levels are stored in the large **DUKE3D.GRP** file.

The floors of the sectors are varied in height to create a pew, altar, and a dais on the altar. The room basically mirrors itself on the north and south sides. The major difference between the two sides is that the ceilings slope in different directions, creating a peaked ceiling. This is a good room to load into Build and study in depth later for a better understanding of using many sectors to create a single unified area. Figure 2.12 shows a 3D view of the pagan chapel.

The primary thing to remember when creating your own levels is that a sector is an enclosed area of space with a separate floor and ceiling. A sector also defines a valid player space. Keep this in mind as you create your structures from now on, and you should be fine.

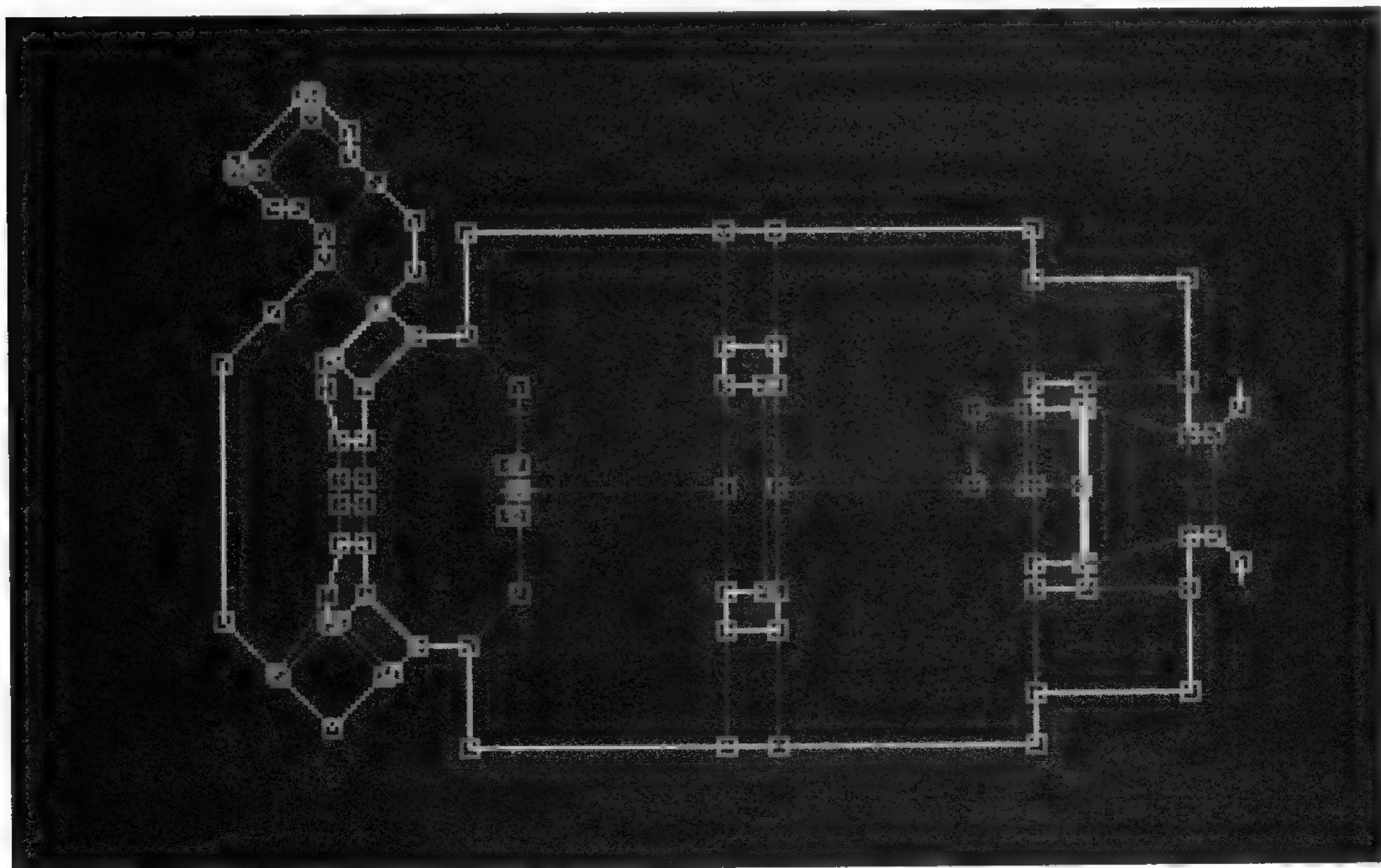


FIGURE 2.11: Note the symmetry in the sectors that make up the pagan chapel in this 2D view.

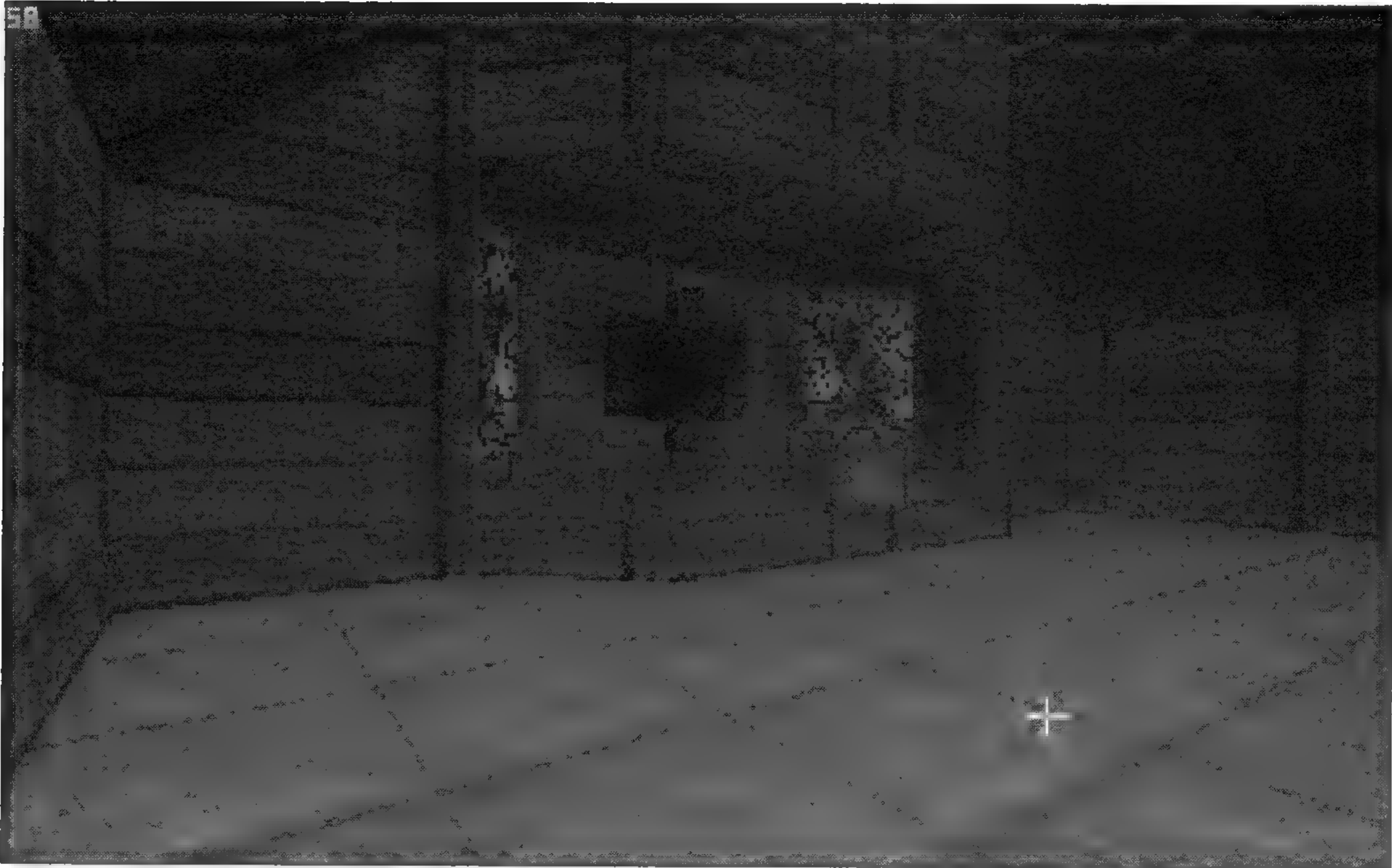


FIGURE 2.12: Here is a 3D view of the pagan chapel.

VUEWING YOUR OWN LEVEL IN 3D

So far there is only one place on your little one-room level that the player can go to on the map: the lone sector. All of the area outside the sector is what's known as *null space*, which means that no player or object can be placed there.

Now it's time to see what your level looks like in the game. One of the best parts about Build is that you don't have to leave it and go into the game itself to see what the level will look like. Instead, as you've already seen, Build has its own 3D view mode. To go into 3D mode, press the Enter key on the *numeric keypad* (the numeric keypad's Enter key is the one that's to the right of the numeric keypad on your keyboard).

When you press the numeric keypad's Enter key, you should see a small room that has a boring brown brick texture on all the walls, the floor, and the ceiling, as shown in Figure 2.13. You can use the arrow keys to move around in your room, just as you would use the arrow keys to move yourself around in the game. Go ahead and move all around your single room. Once you've done that awhile, press the numeric keypad's Enter key again to put yourself back into 2D mode.

You may notice that the white arrow has moved since you last were in 2D mode. The white arrow represents your point of view within the sector. When you switch to

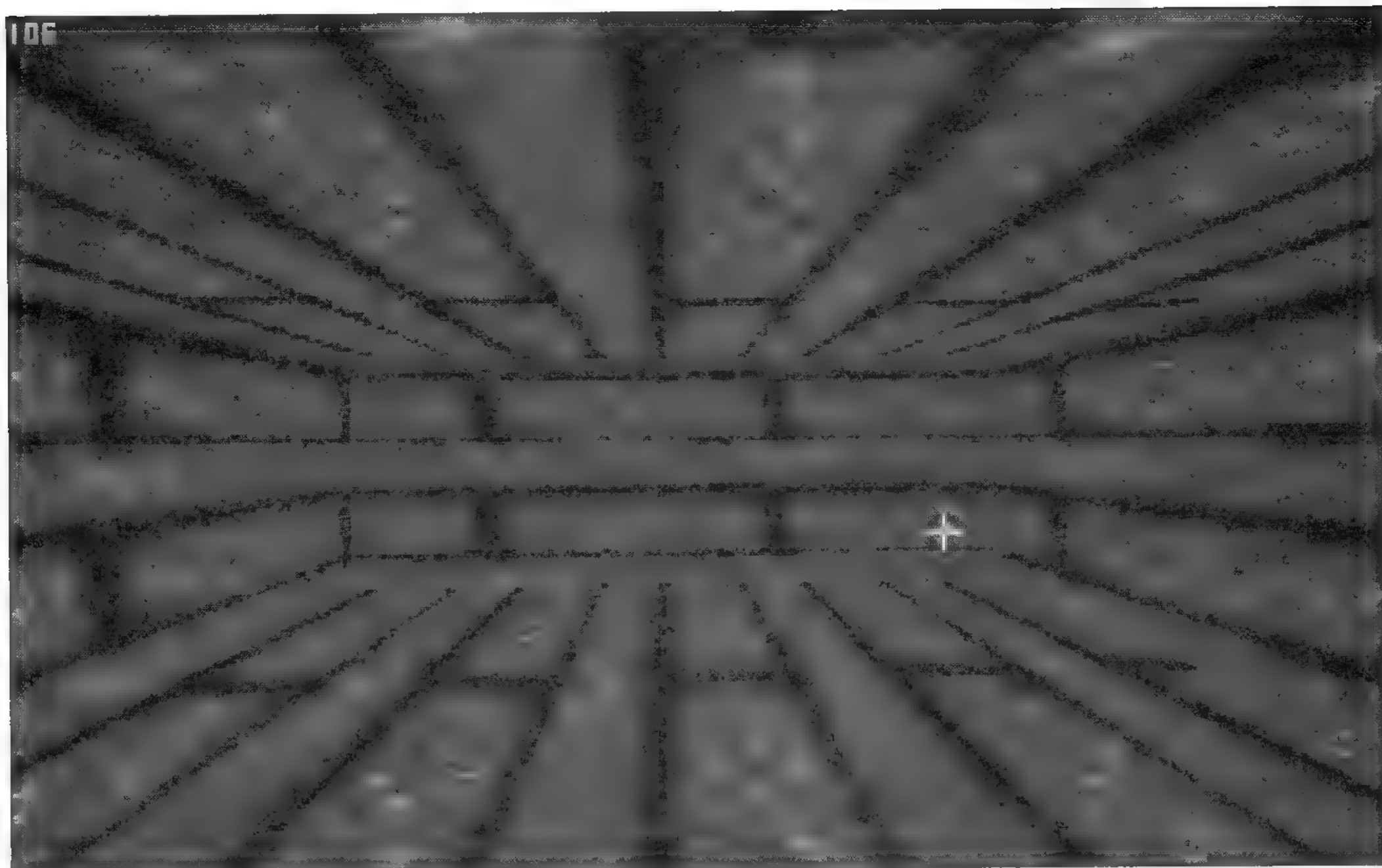


FIGURE 2.13: Your one-room sector should appear similar to this in 3D view mode.

3D mode, you are standing on the spot represented by the white arrow, and you are facing in the direction that the arrow is pointing. Because you moved around in 3D mode, the white arrow is in a different place now that you've come back into 2D mode.

Now, use the same arrow keys that you used in 3D mode to move yourself around. You will find that the white arrow moves on the map in the exact same manner that you would move as a player while in 3D mode. You can use the arrow keys to change the player's point of view in either 2D mode or 3D mode. Notice that while moving the white arrow in 2D mode using the arrow keys, you cannot pass through any of the white lines that make up the boundary of the room. This is because the white lines represent solid walls, and the 2D movement of the white arrow remains consistent with the 3D movement by not allowing you to pass through a solid wall.

Another method to move the white arrow is to place the mouse cursor in a new location on the map and then click the right mouse button. The white arrow will immediately jump to the location of the mouse cursor. Using this method, you will be able to quickly travel from place to place while creating your level, without having to actually travel from one corner of the map to the other by maneuvering from room to room.

While in 2D mode, you will also notice a brown arrow where the white arrow used to originally be located. This brown arrow represents the starting place for the player

in this level during game play. You can move the start position of the player at any time by placing the *white* arrow in the location you would like the player to start and then pressing the Scroll Lock key (located in the upper-right corner of your keyboard, next to the Print Screen key). Pressing the Scroll Lock key moves the brown arrow to the current location of the white arrow. Make sure that the brown arrow is within a valid player space at all times. If you try to test your level in the game and you die immediately upon entering the level, chances are your brown arrow is not in a valid player space. Unfortunately, Build will not warn you of this condition, so it's something you'll have to look for yourself.

SAVING YOUR OWN 3D LEVEL

Congratulations, you just made your first *Duke Nukem 3D* level! Not that it will win any contests or anything, but it is a valid level that will work in the game. Before you continue, go ahead and prove to yourself that it does indeed work in the game. While in 2D mode, press the Esc key. In the status bar, you will be presented with a menu of the following command choices:

(N)ew, (L)oad, (S)ave,
save (A)s, (Q)uit

Press the S key to save the map. The map will automatically be saved with the file name NEWBOARD.MAP. Press Esc again to bring back the menu, and press Q for Quit. Be sure to press Y to answer yes to the obligatory "Are you sure?" question, and finally, press N to respond that you do not want to save changes. (You just did save the map, so there's no need to do so again.) You should now be back at the DOS prompt, still inside the main game directory. If you now display a directory (with the DIR command), you should see the new file NEWBOARD.MAP. This is the level that you just created. To play this map in the game, type the following command on the DOS command line:

DUKE3D MAP NEWBOARD



NOTE

If you get the error message "Arrow must be inside a sector before entering 3D mode." when trying to switch from 2D mode to 3D mode, this means that the white arrow is not within a valid player space. Instead, it is somewhere in null space. With the simple one-sector level that you've created so far, you only have one place to put the white arrow so it's within a valid player space—inside your single sector. To move the white arrow into the sector, move the mouse cursor so it's inside the sector and then click the right mouse button. The white arrow will jump to the location of the mouse cursor. Once the white arrow is within a sector, you will be able to enter 3D mode.

This is the command to run an external MAP file with the game. Note that the MAP

extension is not necessary. *Duke Nukem 3D* should start, and after initialization you should see the message “Entering User Map NEWBOARD.MAP.” After a few more seconds of loading, you’ll be playing your very first level. It’s a great feeling to actually create something for the first time and to see it come to life in a game. Alas, in this case, that feeling is fleeting because your first level is the most boring of all possible levels and there’s not much to do with it at this point. However, the point here is to show yourself that you were able to successfully create, save, and load your first *Duke Nukem 3D* level. Now that you’ve done this, just exit from the game. Next, go on to chapter 3 so you can work on turning this boring little level into a more interesting one.



NOTE

You might have noticed that when you were playing your sample level, *Duke Nukem 3D* skipped all the title screens and menus and instead jumped immediately into your map. The title screens and such were intentionally programmed to be skipped when play-testing an external level. This helps you save time when entering the game for this reason. In the long run, this will save you a lot of time, believe me.

KEYSTROKE AND OPERATION SUMMARY

A summary of all of the keystroke sequences and operations covered in this chapter appears in Table 2.1.

TABLE 2.1: CHAPTER 2'S KEYSTROKES AND OPERATIONS

| KEY SEQUENCE/OPERATION | FUNCTION | VALID VIEW MODE |
|---|--|-----------------|
| Spacebar | Start sector drawing mode | 2D |
| Spacebar (while in sector drawing mode) | Create a vertex | 2D |
| Backspace | Delete last vertex created | 2D |
| a) Spacebar to create a vertex on top of an existing vertex of the current sector, or b) Backspace to delete the first vertex created in sector drawing mode | End sector drawing mode | 2D |
| Enter (numeric keypad) | Toggle between 2D and 3D view modes | 2D/3D |
| Arrow keys | Change viewpoint position (move the white arrow) | 2D/3D |

| KEY SEQUENCE/OPERATION | FUNCTION | VALID VIEW MODE |
|------------------------------|--|-----------------|
| Scroll Lock | Change player one start position to current white arrow location | 2D |
| Esc, S, | Save a level | 2D |
| Esc, Y, Y | Save a level | 3D |
| Esc, L, choose level, Enter | Load a level | 2D |
| Esc, A, type new name, Enter | Save level with a new name | 2D |
| Esc, N, Y | Start a new level | 2D |
| Esc, Q, Y, Y | Quit Build | 2D |





Creating Sector Groups

In the previous chapters you were introduced to *Duke Nukem 3D* game elements, you created your first real level consisting of one whole sector, and you studied several example areas from the first episode of *Duke Nukem 3D*. As you saw, these examples illustrated exactly what a sector is and how many sectors can be combined to form complicated architectural structures. Now it's time for you to try to learn exactly how to create groups of sectors, so you can start building these kinds of structures.

RETURNING TO BUILD

If you worked through chapter 2, you have by now tried out your first level in *Duke Nukem 3D* and should now be at the DOS prompt. To return to Build, type **BUILD** and press Enter. This time, notice that Build automatically brings up your level and places you in 3D view mode to start. You might recall that this is different than the first time you started Build. That time, Build started in 2D mode with an empty map. However, if you are working on a level and leave it stored in the file with the default name NEW-BOARD.MAP, Build will automatically load that map each time you start the program. Alternately, you can load any level that you want by simply typing the name of the level at the command prompt, such as in the following example:

```
BUILD MYLEVEL
```

This example assumes there is a level map named MYLEVEL.MAP in the main Duke directory.

MODIFYING A SECTOR

Your first exercise in modifying your sector will be changing its shape. Although you began by making a rectangle, sectors don't have to be rectangular at all. As you have seen if you have experienced playing the game, sectors can have just about any shape imaginable, as long as the lines that make up a sector don't cross.

To alter the shape of a sector, you must be in 2D view mode. Press the numeric keypad's Enter key to switch to this mode now. You should be presented with the 2D view of your single sector, just as you left it. Before you modify your sector, though, now is a good time to make some more room on your map so that you will have more area visible on the screen. This is done by *zooming* the 2D view out.

ZOOMING YOUR VIEW

The 2D map can be zoomed in either direction by using the A and Z keys. The A key zooms the map in (making all the structures look larger), and the Z key zooms the map out (making all the structures look smaller). When you use these keys, you're actually not changing the size of the structures on the map—you're just getting closer to or farther from the map's surface. Zooming in (pressing A) provides you with a more detailed view of a portion of your work; zooming out (pressing Z) provides you with a more global view of the entire level. Once you've zoomed out a bit to give yourself some more room, you'll be ready to alter the shape of the sector.

CHANGING THE SHAPE OF A SECTOR

Changing a sector's shape is done by placing the mouse cursor on one of the *vertices* and moving it around. If you'll recall, the vertices are the points at which two of the walls meet. They are represented on the 2D map by a small green square. Place the cursor on any one of the vertices, click and hold the left mouse button, and then move the mouse. The vertex on which you click should move with the mouse (it will also snap to the gray grid lines). When you have the vertex where you would like to place it, simply release the mouse button. The walls attached to the vertex you move should also move, and your sector is now a different shape.

Practice moving any or all of the four vertices that make up your sector, until you have a sector that's an irregular shape. An example is shown in Figure 3.1. Once

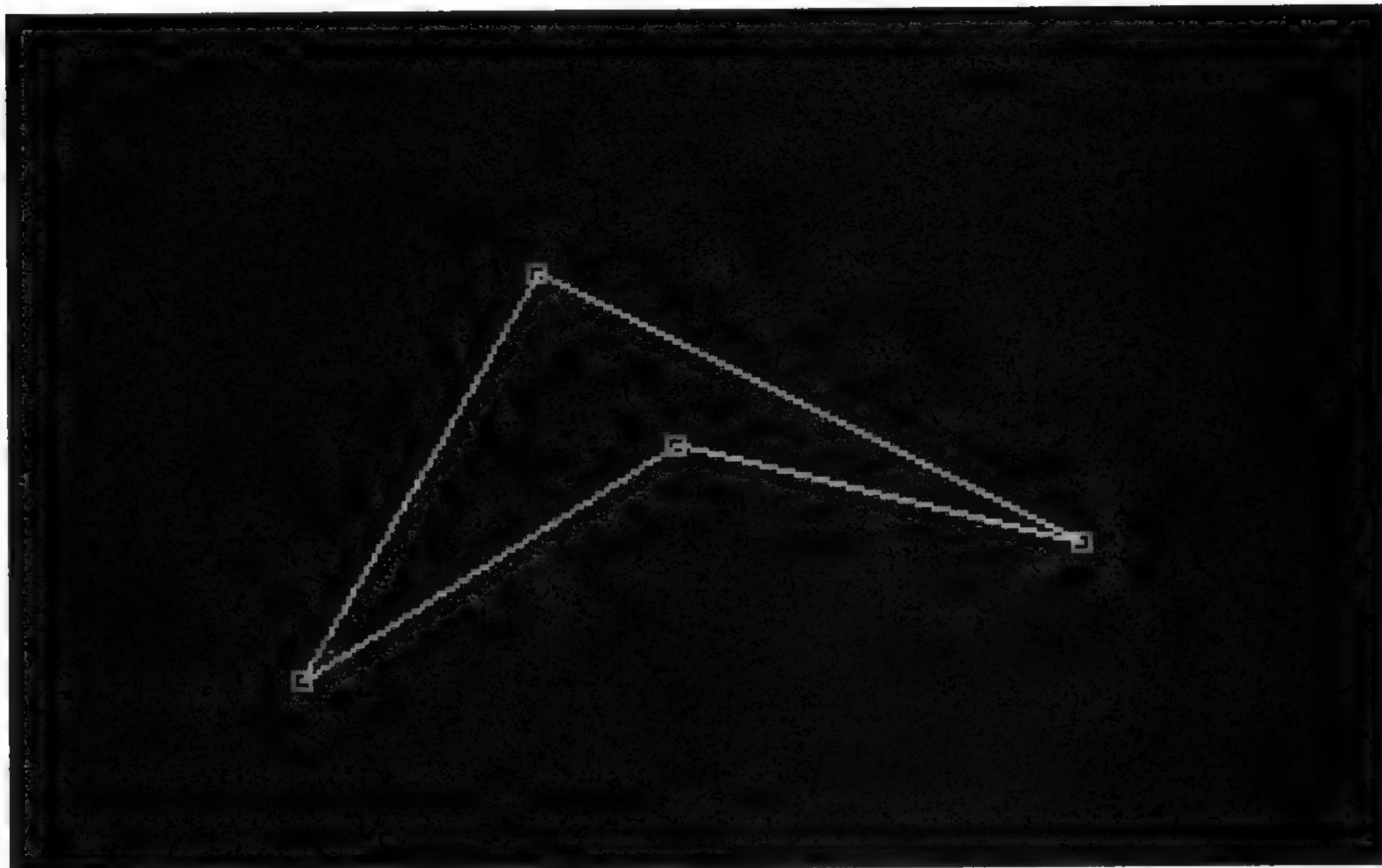


FIGURE 3.1: Here is a sector with an irregular shape.



TIP

Recall that you can move the white arrow inside a sector quickly by right-clicking on the spot you would like the arrow to be.

you have an odd-shaped sector, flip back to 3D mode momentarily to verify that it indeed matches the shape you drew in 2D mode. (Make sure the white arrow is still inside the sector before you try to flip to 3D mode, or Build will give you the error “Arrow must be inside a sector before entering 3D mode.”)

After you check out the 3D version of your sector, return to 2D view mode. As a final exercise in moving vertices, move the four vertices back into a rectangular shape once again.

For the next exercise, you might want to make the rectangle a little larger than it was originally.

SPLITTING WALLS

Your original sector was created with four vertices and four walls. Sectors are not limited to having only four sides, however. You can make very complicated sectors with many walls and many vertices.

Just like you can change the shape of a sector after it's created by moving the vertices around, you can also change the number of sides a sector has by splitting a wall. When you split a wall, you take a single wall and turn it into two walls by inserting a vertex somewhere along that wall. Your sector will then have one more wall and one more vertex than it did originally. This new vertex will behave exactly like all the original ones, which means that you will be able to drag it wherever you like to change the shape of the sector.

Splitting a wall is done in 2D mode. To do so, place the mouse cursor on the wall in the place you would like the new vertex and press the Ins key. This will split the wall into two walls and add a vertex in the mouse cursor's location. This new vertex will join the two new walls. Try splitting the northern wall of your rectangular sector by placing the mouse cursor anywhere on that wall and pressing the Ins key. Your map should look similar to the one shown in Figure 3.2.

After splitting the north wall, you now have a sector with five walls instead of four. This allows you to create a pentagon-shaped room if you wish, or another five-sided shape. For now, just leave the room in its current shape.

The next thing you can do is split one of the now two north walls *again*, which will create a third wall on the northern side of your sector. Do this now by placing the mouse cursor on either of the two walls and pressing the Ins key. After you have done this, move the two vertices on the north side of the sector so that the three walls that make up the north side of the room are roughly the same length, as shown in Figure 3.3.

CREATING A NEW SECTOR

Now it's time to expand your level a bit by creating a second sector. If you'll recall, sector drawing is done by pressing the spacebar when the mouse cursor is in each spot

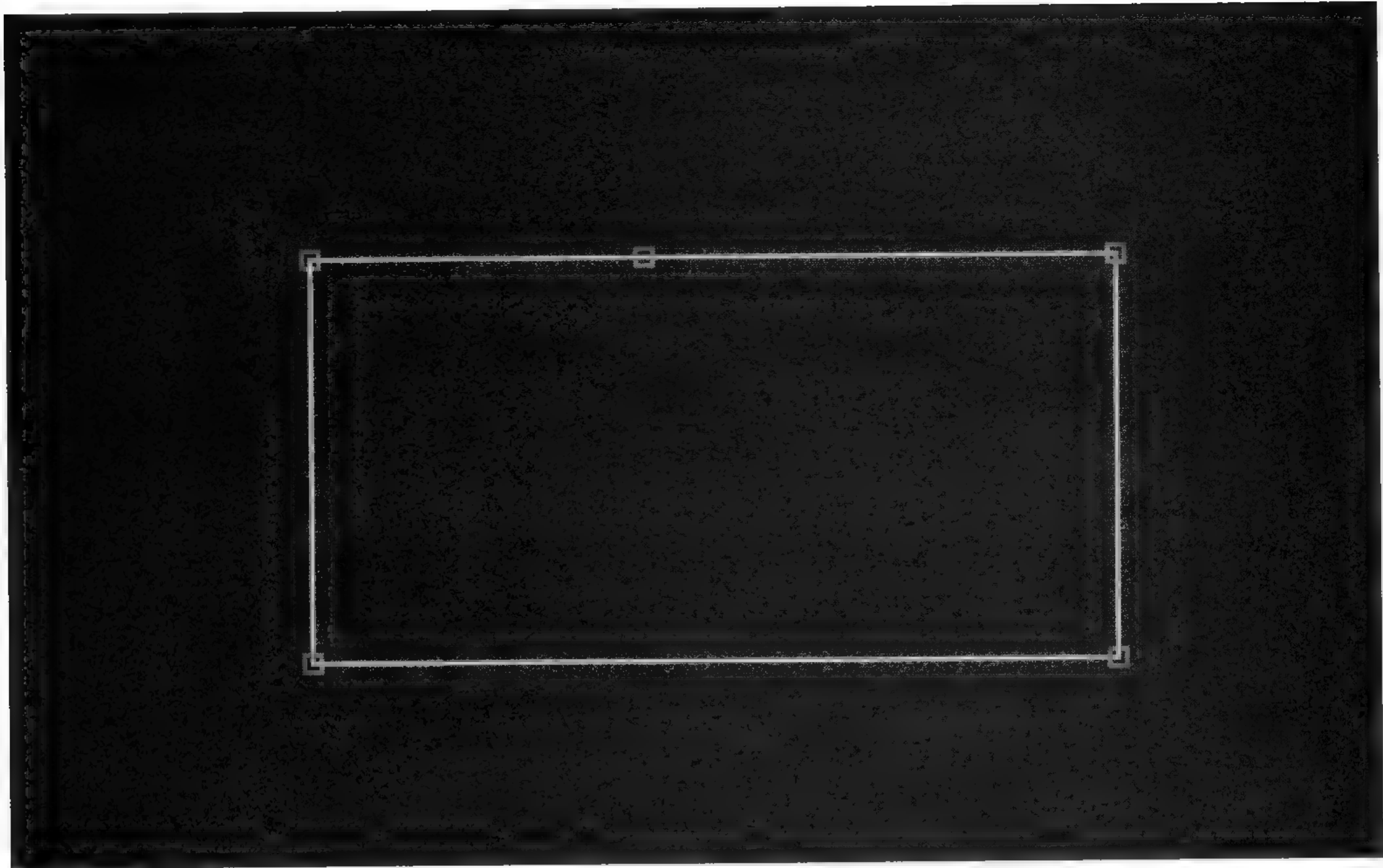


FIGURE 3.2: After you split the north wall by adding a vertex, the sector now has two north walls.

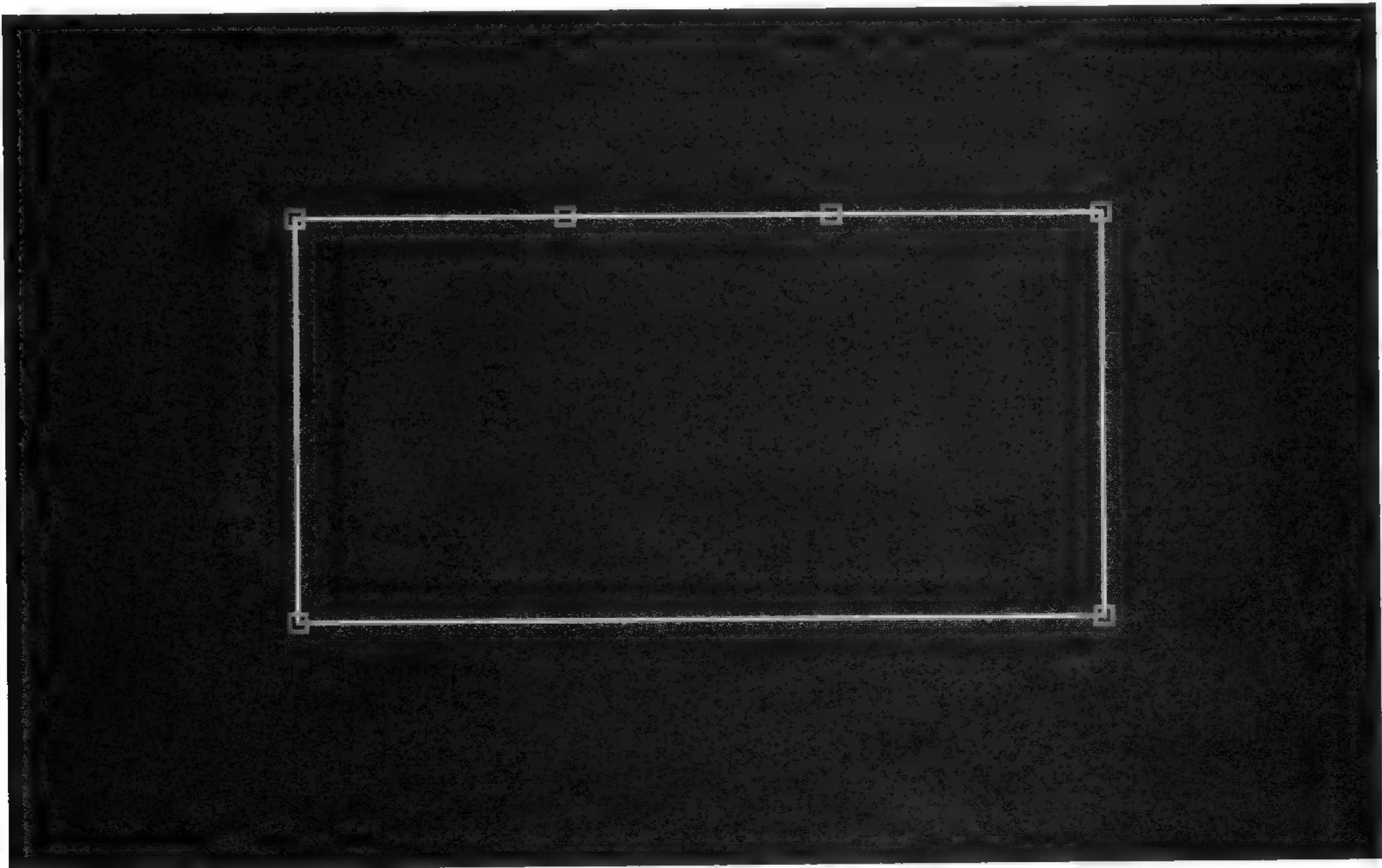


FIGURE 3.3: After adding another vertex to one of the north walls and moving the vertices around a bit, you should have three walls of equal length on the north side of the room.

where you would like a new vertex. The sector becomes complete when you place a vertex on the spot of the first vertex, creating a closed path.

For the next operation you are going to place the second sector so it's connected to the first sector. This will enable you to easily pass between the two sectors. The completed product will look similar to the level shown in Figure 3.4. Note that the new sector also contains four vertices. Two of these vertices are new (the northern two), but the southern two vertices already existed; they are the two vertices that you created in the previous section by splitting the north wall of the first sector. For the purposes of the next example, I will refer to these two vertices as the *new west vertex* and the *new east vertex*.

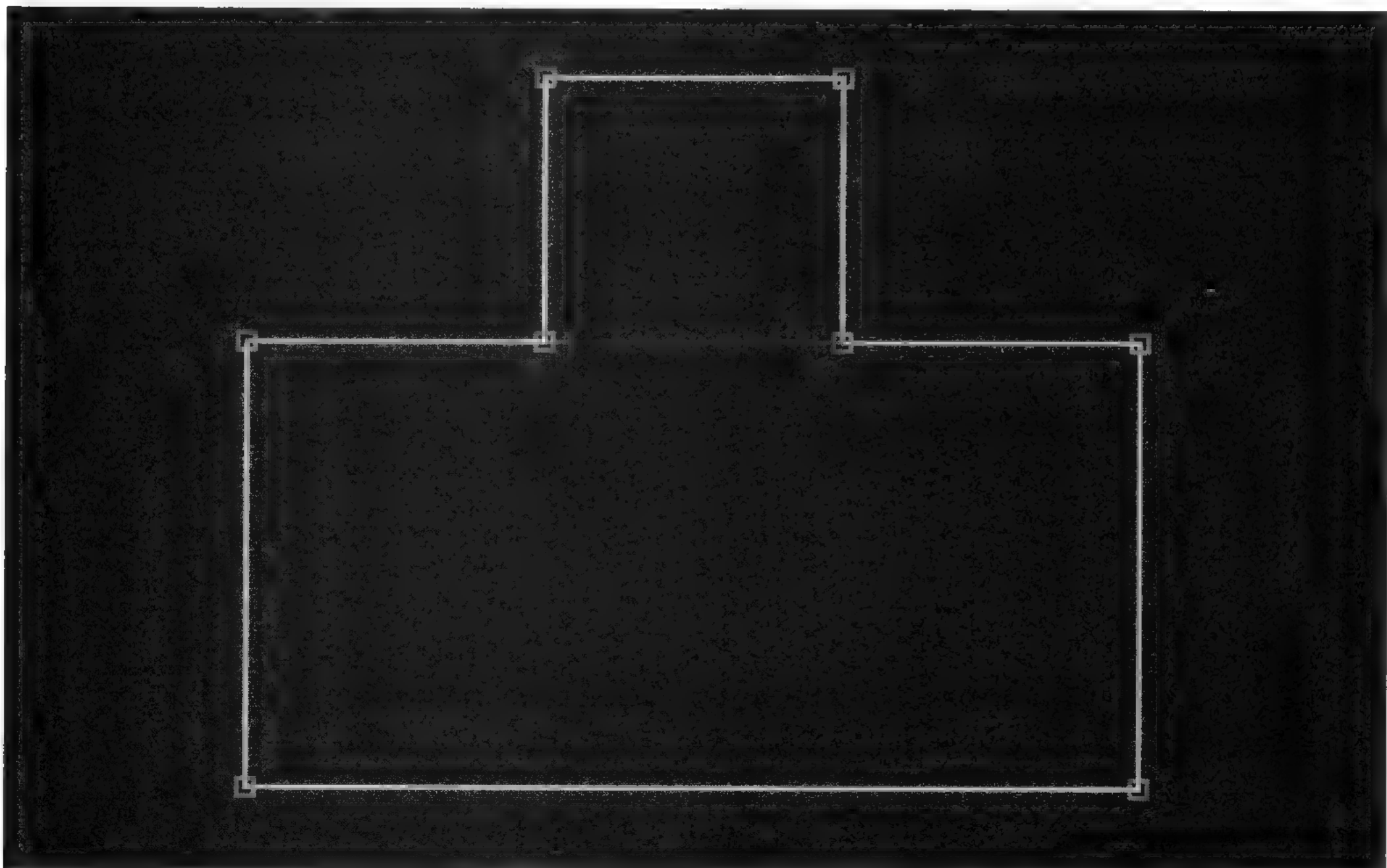


FIGURE 3.4: Adding another sector to your first sector will create a level similar to this one.



Even though the sector you are about to create has two of its four vertices created already, you are not going to do anything different to create this sector. In fact, you will go through the exact same steps to create this sector that you did to create the first one. Carefully execute the following steps to create sector number two:

1. Place the mouse cursor several gridlines to the north, and directly above, the new west vertex, and press the spacebar. You will see the message “Sector drawing started.” on the status bar.
2. Move the mouse cursor due east until it is directly north of the new east vertex. You’ll notice that a white line will be anchored at the vertex you set when you pressed the spacebar. When the mouse cursor is due north of the new east vertex, press the spacebar.
3. Place the mouse cursor *directly on* the new east vertex, and press the spacebar.
4. Place the mouse cursor *directly on* the new west vertex, and press the spacebar.
5. Place the mouse cursor *directly on* the first vertex that you drew in step 1, and press the spacebar. You should see the line between the new east vertex and the new west vertex change color from white to red. Your sector should be complete and look similar to the one shown in Figure 3.4.

You have just created a second sector, directly attached to the first one. Note that the wall attaching the two sectors is now red instead of white. A red wall in 2D view mode is said to be *transparent*. This is where the definition can get a bit confusing, so bear with me here. When you read the word *wall*, you normally think of the flat solid structures that separate rooms. Those solid objects are indeed walls in the Build world, but a *wall* can also mean any boundary between sectors, even if that boundary is transparent. So, *walls* are either solid and shown in white or transparent and shown in red.

If you’re still confused by this, perhaps it will make more sense to go into 3D mode. When you go into 3D mode, you should see what appears to be a single T-shaped sector. You know, however, that there are actually two sectors present. Try to use the 3D view to find the boundary of the two sectors (it might help to toggle back and forth between 2D and 3D modes to see where you are and which way you’re facing). Figure 3.5 shows a 3D view of the two-sector level, facing the boundary between the two sectors.

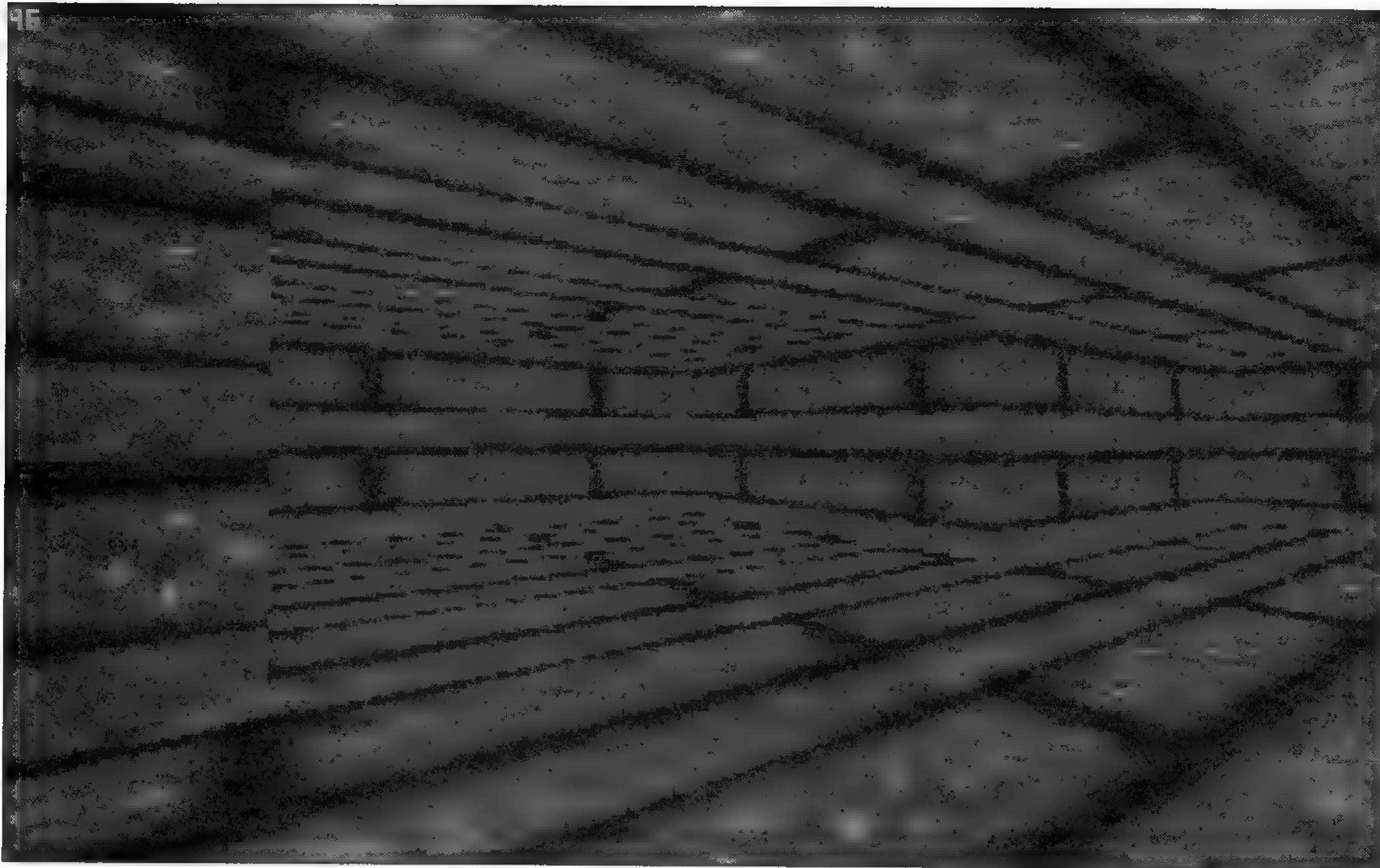


FIGURE 3.5: Here is a 3D view of the transparent boundary between the two sectors.

When you find the boundary between the two sectors, notice that there's no visible indication of any boundary there. This is why we call the wall between the two sectors *transparent*, simply because you can't see it. In fact, at the present time, you can't tell if this is two sectors or simply one large T-shaped one.

A transparent wall is *two-sided*: there is valid player space on each side of the wall. Note that all of the white walls you've drawn only have player space on one side of the wall; the other side of the wall is null space. The white walls are referred to as *one-sided* walls. Figure 3.6 illustrates this. Note that the first sector you drew is marked "A," the second sector "B," and the rest of the map "N" for null space. The wall that separates sector A from B is red because valid player space (a sector) exists on both sides of that wall. Walls that border a sector and null space are one-sided, hence white.

The concept of the transparent or two-sided wall is an extremely important one to understand if you're to become an expert *Duke Nukem 3D* level creator.

ADJUSTING FLOOR AND CEILING HEIGHTS

As you can see, there's currently no way to distinguish your two sectors from each other. This is because all of the attributes of the two sectors are identical. Remember

**TIP**

When you hold down the left mouse button while using the PgUp or PgDn key, it locks in the floor or ceiling that you select. As you move the floor, the shape of the sector changes and might result in a new surface moving into place under the position of the mouse cursor. If you do not hold down the mouse button, the object under the mouse cursor each time you press PgUp or PgDn is the object that is affected. Remember to hold down the mouse button when changing floor and ceiling heights, so you do not inadvertently alter a surface.

the definition of a sector: It's an enclosed area of space with a common floor and ceiling. If you study your level as it is now in 3D mode, you'll see that your two sectors have identical floor heights and ceiling heights (not to mention identical textures). If you were to change the floor or ceiling height of one of the two sectors, it would become easier to see where one ends and another begins.

Changing the heights of floors and ceilings is done in 3D view mode. Switch to 3D mode now if you're not there already. Position yourself so you're standing in the first sector, but looking in the direction of the second, smaller sector (see Figure 3.5). Begin by raising the floor height of the second sector.

To do this, place the mouse cursor so it is over the floor of the second sector. Once it is on the floor, hold down the left mouse button, and press the PgUp key one or two times. Each time that you press PgUp, the floor of the sector will

rise a bit. If you go too far, you can press the PgDn key to lower the level of the floor. During the time that you're adjusting the floor with PgUp or PgDn, you should be holding down the left mouse button.

Ceiling heights are altered in the same way. To adjust the ceiling height of a sector, place the mouse cursor so it's resting on the ceiling, hold down the left mouse button, and press PgUp to raise it or PgDn to lower it. For now, lower the ceiling of the second sector one or two *clicks* (that is, press the PgDn key one or two times).

Your sector should now look very similar to the one shown in Figure 3.7. Note that it is now possible to see the border between the two sectors because the difference in the floor creates a little *step*. The texture of this step is mapped along the two-sided wall. Likewise, the little *ledge* that occurs because of the different ceiling heights is also apparent. Its texture is along the two-sided wall. So, even though that wall is known as transparent, there still can be a texture on part of it. These textures can be used to create a step or a ledge that results from differing floor or ceiling heights.

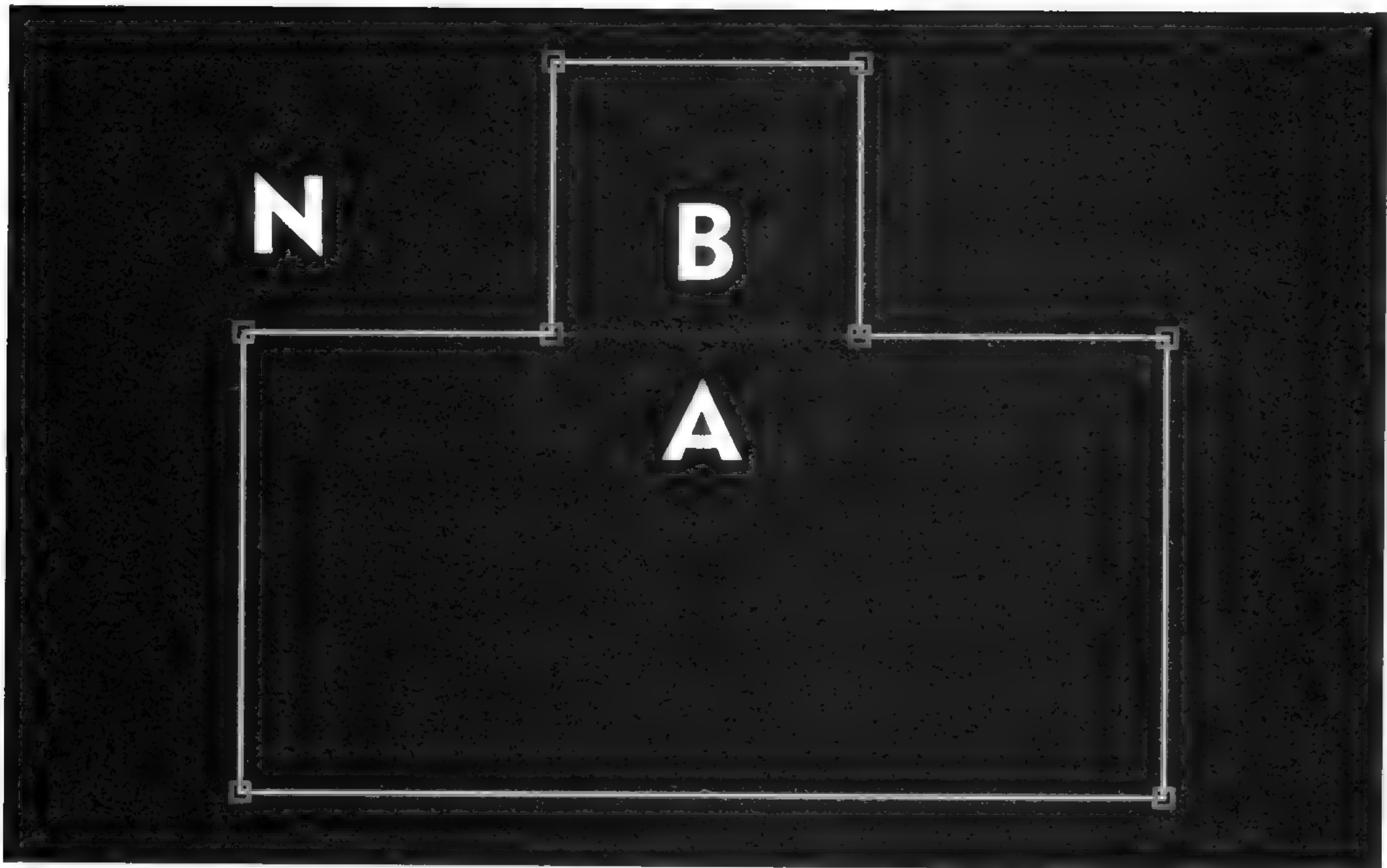


FIGURE 3.6: The red wall borders valid player spaces A and B, so it is referred to as a two-sided wall. The white walls border valid player space on one side and null space on the other side (N), so they are one-sided walls.

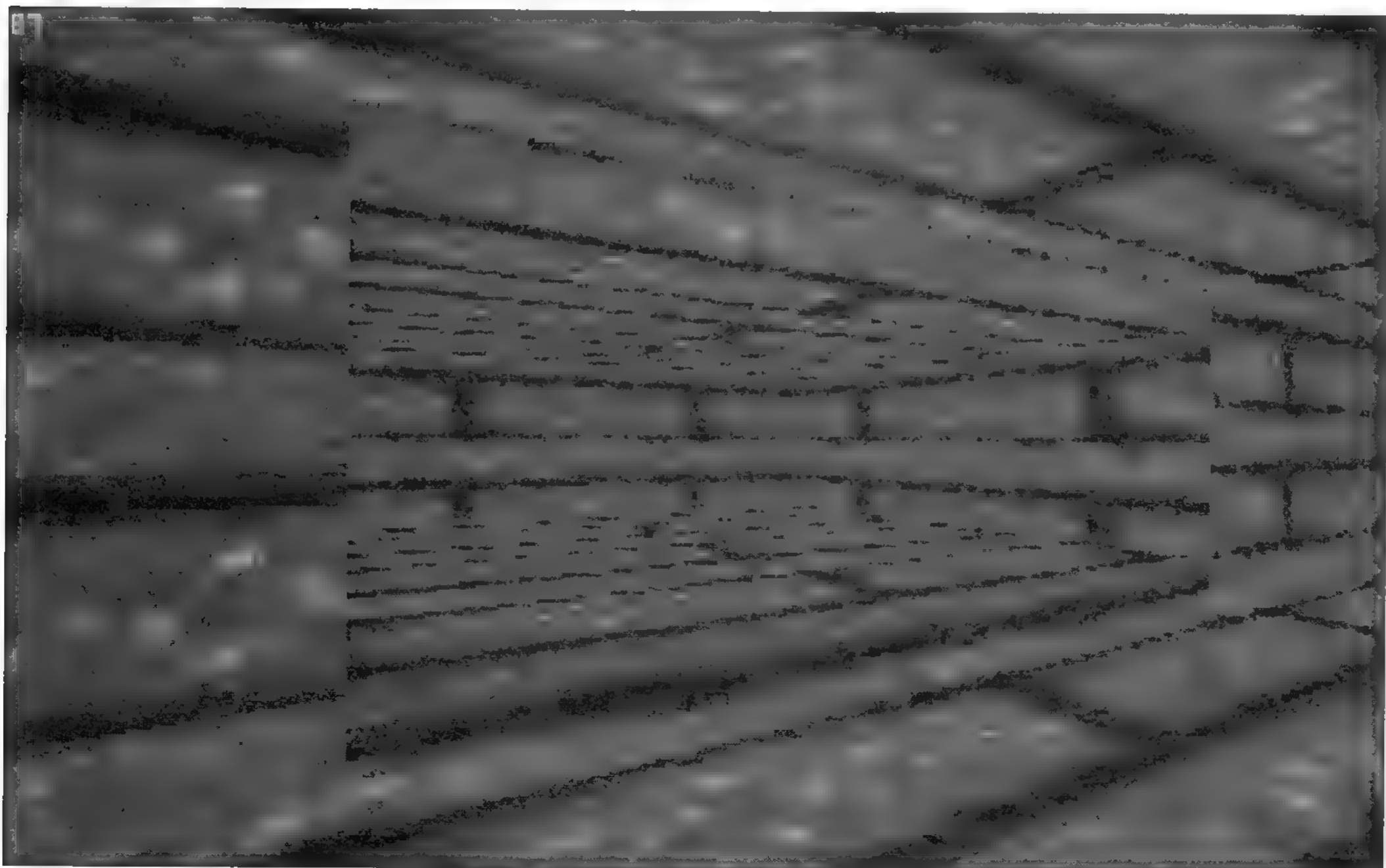


FIGURE 3.7: The second sector now has a lowered ceiling and a raised floor.



SPLITTING ONE SECTOR INTO TWO

Suppose that while creating your map, you form a large room out of a single sector, populate it with monsters, apply a variety of textures, and basically get everything just perfect. After looking at the map a bit more, you decide that you would like to add a small step right in the center of the room. Normally, this would be a difficult task to accomplish. A step could be defined as the difference between two floors, but the imaginary large room just described is made up of a single sector. Because a single sector can only have one floor height, your only recourse is to somehow turn this single sector into two. There are basically two ways to do this.

The first way would be to move all the vertices of the single sector around, effectively shrinking the sector, and then add the new sector in such a way that the border between the old sector and the new one becomes the step. This method can be difficult because you're required to change the shape of the first sector dramatically before adding the new one, and I've already stated that you've got the large room *exactly* the way you want it (except for the damn step!).

Fortunately, Build provides you a second way that's more convenient in this situation. You can divide an existing sector into two, and each becomes its own separate sector, which can be edited separately. If you were to split the large room example into two sectors, you would preserve the original shape of the room, but still get your step.

Now go ahead and split your first sector along the east and west walls, so you create two thin north and south sectors, by following these steps:

1. Start off in 2D view mode.
2. Split the west wall of the first sector by inserting a vertex along that wall. Recall that to do this you place the mouse cursor somewhere on the wall you want to split and press the Ins key.
3. Split the east wall of the first sector in a similar manner.
4. Move the two new vertices so that they split the length of the original east and west walls of this sector evenly, by placing them halfway between the northern and southern vertices of this sector.

5. Place the mouse cursor on the western vertex that you just created, and press the spacebar. The message “Sector drawing started.” should appear on the status bar.
6. Move the mouse cursor to the eastern vertex that you just created, and press the spacebar. The message “Sector split.” should now appear on the status bar, and the sector drawing process should be complete. Your final product should look similar to the one shown in Figure 3.8.

That’s all there is to splitting a sector. Note that Build accounted for the line you drew connecting two opposite walls of an existing sector, so it deduced your intention and split the sector automatically. It also turned that new line into a two-sided (red) wall automatically, because it borders two sectors—the two sectors you just made from one. Now that your first sector is split in two, you can edit the sectors’ ceiling and floor heights separately.

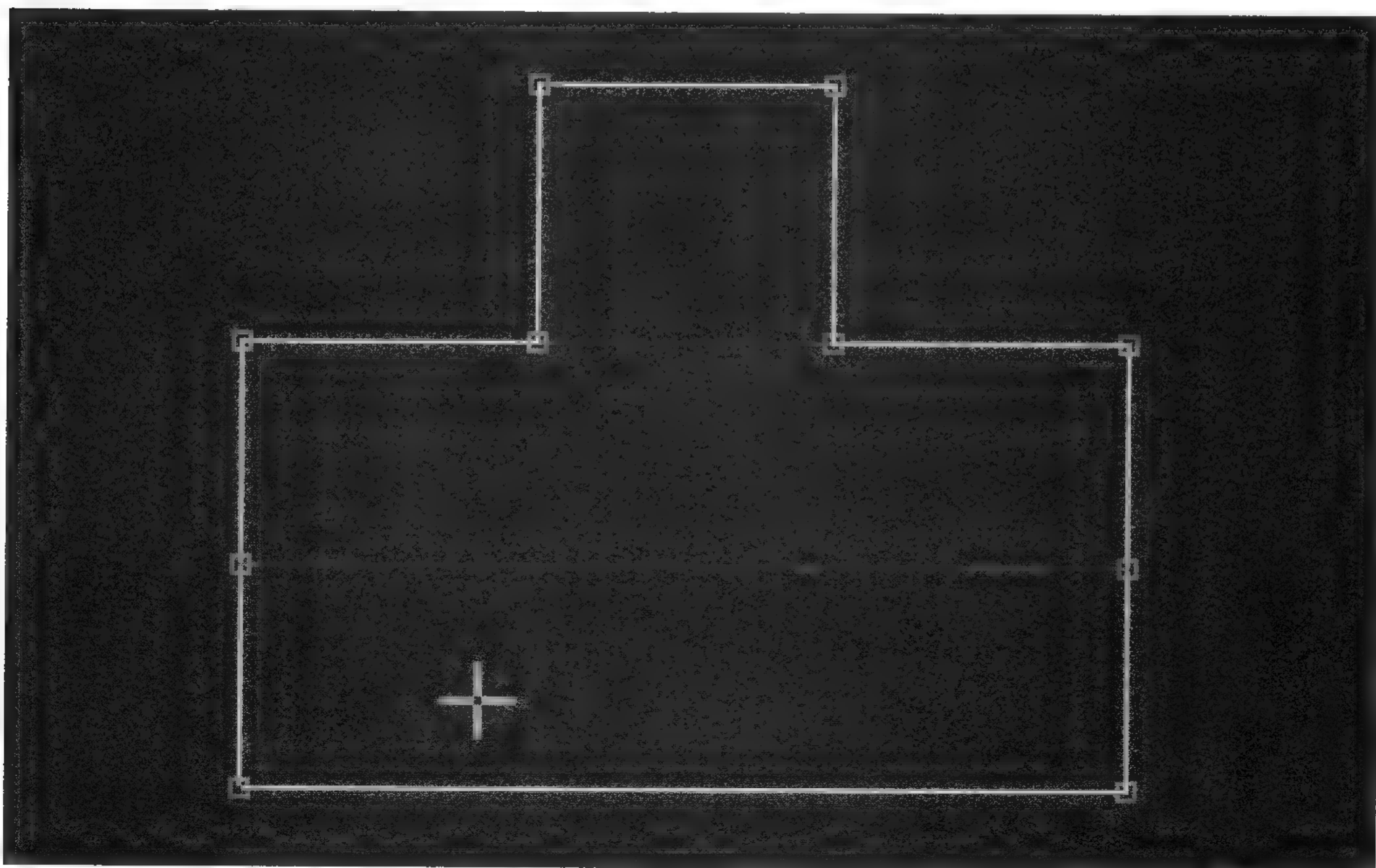


FIGURE 3.8: To split a sector in two, split two opposite walls with vertices and draw a single line between them. Build splits the sector automatically.

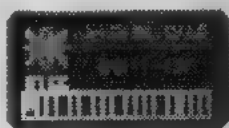
JOINING TWO SECTORS INTO ONE

Build also lets you join two separate sectors into a single larger one. For example, suppose that after looking at that large room described previously, you decide that you don't want the step you created after all, and you'd rather combine the two split sectors back into the original one sector.

For practice you'll rejoin the two small sectors you just created in the last exercise back into the original larger sector. Joining two sectors is very simple; follow these steps:

1. Start off in 2D view mode.
2. Place the mouse cursor into one of the two sectors that you wish to join. Press the J key.
3. Move the mouse cursor into the second of the two sectors that you wish to join. Press the J key again.

Your two sectors join and once again become a single sector. Note that this join-sectors operation can be done for any two sectors on your map that share a wall; the two sectors do not have to have once been a single sector that was split as in this example. As long as two sectors are joined by a red (two-sided) wall, they can be joined into one sector. This has the effect of removing the two-sided line that bordered the two sectors.



NOTE

If the two sectors that you join have different attributes, such as different ceiling or floor textures or different heights, the new joined sector will retain the attributes of the first sector you select with the J key, and the attributes of the second sector will be discarded.

DELETING VERTICES

Your sector is almost back to the way it looked when you first created it—with one exception. The east and west walls of the room are still split into two walls. To get your sector back to its original shape, you can delete the two vertices that split the walls.

Deleting a vertex is done by placing the mouse cursor on the vertex that you want to delete, pressing and holding the left mouse button, and dragging that vertex to another vertex

that's right next to the one you want to delete (a vertex on the same wall). Once the vertex is directly over the second vertex, release the mouse button, and Build will delete the desired vertex. In this case, dragging the west central vertex and dropping it on the vertex directly south of it will delete it, and likewise dragging the east central vertex and dropping it on the vertex just south of it will delete it. (See Figure 3.9.) Once your two sectors have been joined into one and the central vertices on the east and west walls are properly deleted, your level should look like it did in Figure 3.6.

CREATING A CHILD SECTOR

Many times it's desirable to create a sector that lies completely inside another sector. Such sectors are called *child* sectors, and the sectors they lie within are naturally referred to as *parent* sectors. For example, a child sector with a lower floor than its parent forms the basis for a pit in a floor. Likewise, a child sector with a *higher* floor than its parent forms the basis for an altar or a table. On your level map you can make a

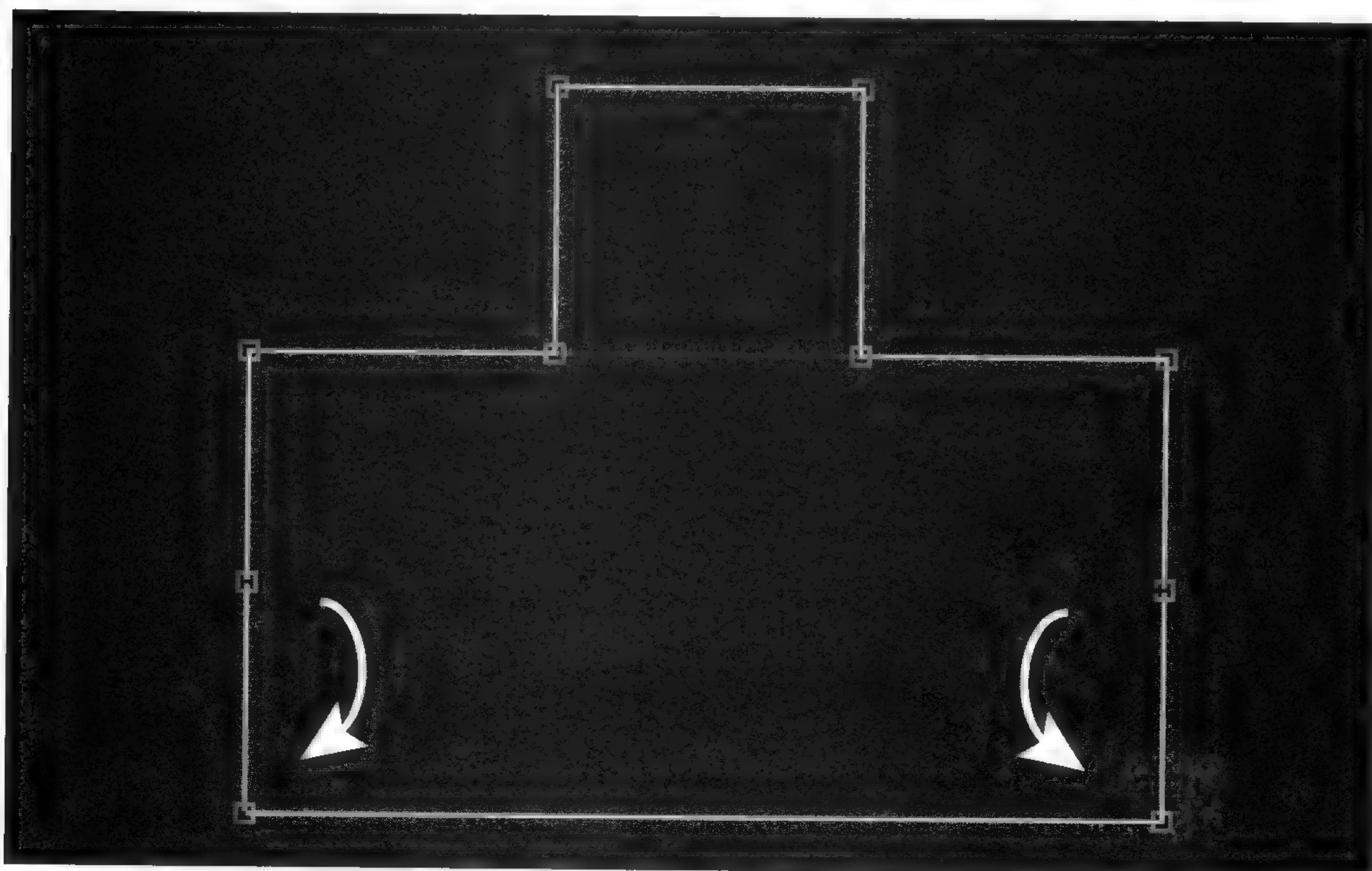


FIGURE 3.9: Deleting the two vertices that split the west and east walls is done by dragging each to the vertex just south of it and dropping it on that vertex.

child sector that lies within your first sector, and you can raise the floor of the child sector to turn it into a platform.

You begin a child sector the same way that you create any sector. To begin, place the mouse cursor in the upper-right corner of the first sector, a few grid lines both to the left and below the upper-right corner vertex. When the mouse is at this location, press the spacebar to start the sector drawing process. Draw the child sector the same way that you've already drawn other sectors from scratch, by moving the mouse cursor to the four corners of a rectangle and pressing the spacebar to set each vertex. This time, however, the new rectangle should lie completely inside the first sector, as you can see in Figure 3.10.

Notice that all of the walls of the child sector are white. If you now go into 3D mode and look at the new rectangle (child sector), you'll notice that the walls you just created are solid and that they form a column in the room. This is a great structure as it is, if the intended goal is to create a solid column. However, suppose you want to create a platform.

Recall that the definition of a white (one-sided) wall is a wall that acts as the border between valid player space and null space. This holds true for the area you've just drawn.

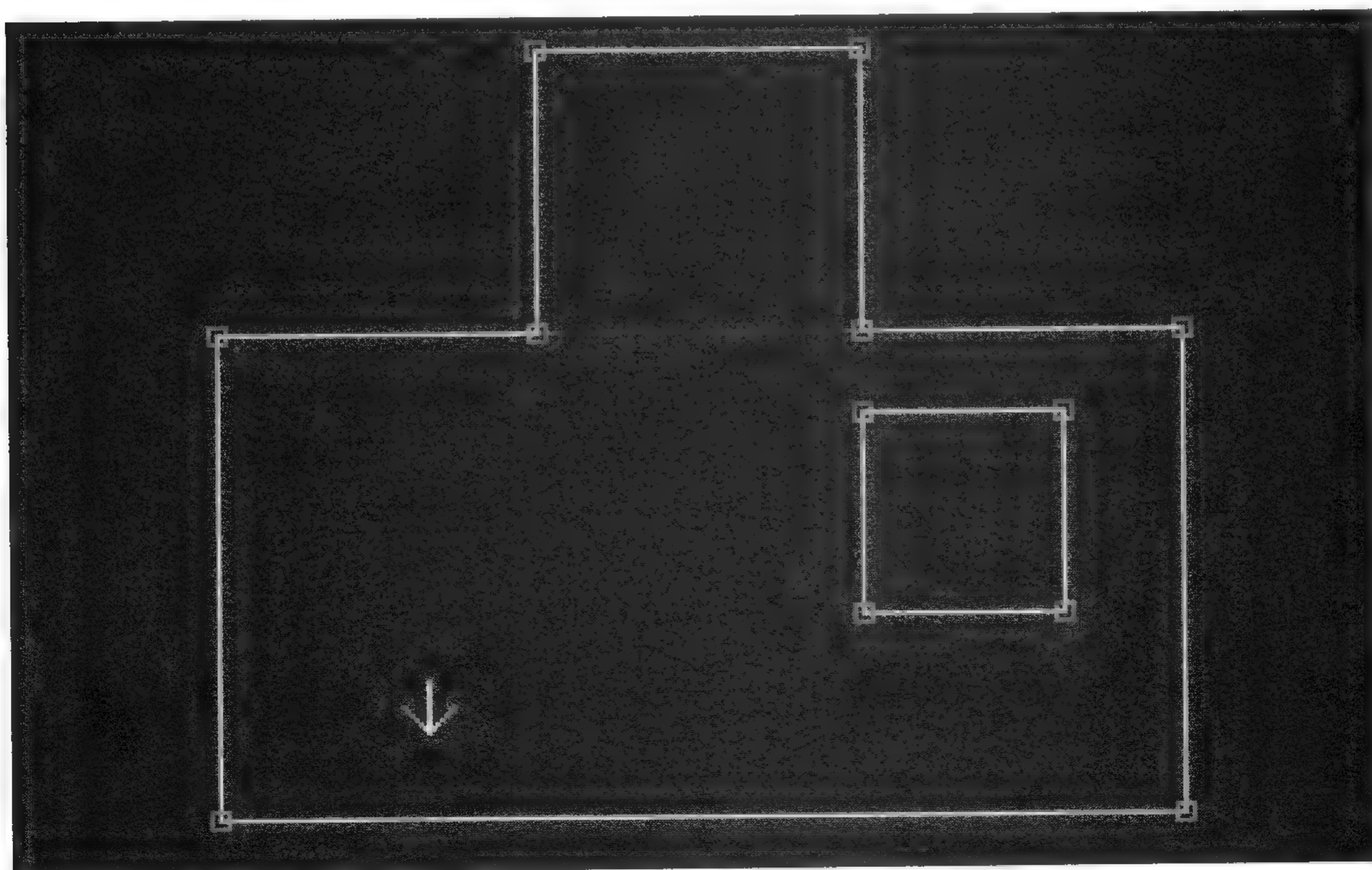


FIGURE 3.10: The first step in creating a child sector is to create a rectangle inside the intended parent sector.

In its current form this new rectangle is not truly a sector yet because the area enclosed in the rectangle is not valid player space, it is *actually* null space. This is why the walls of the new rectangle are white: They separate your first sector's valid player space from the null space within. What you've done is carve out a little square of null space within your first sector. This makes sense with the 3D view of the map, as you can see. The new rectangle consists of solid walls that form a column. The walls are solid because there's null space on their other sides.

This concept can be tricky to understand. As you can see in Figure 3.11, a rectangle created inside a sector starts off as an area surrounding null space. In effect, it sections off an area of null space from within the parent sector's valid player space.

To turn a newly drawn rectangle inside a sector into a valid child sector, return to 2D mode, place the mouse cursor inside the rectangle, and press Alt + S. You should see the white walls of the rectangle turn red, and the message "Inner loop made into new sector." should appear on the status bar. The result should look similar to the one shown in Figure 3.12.

Now that your new child sector is created, go into 3D mode to check it out. The first thing you'll notice is you can't see it! Why not? You can't see the child sector

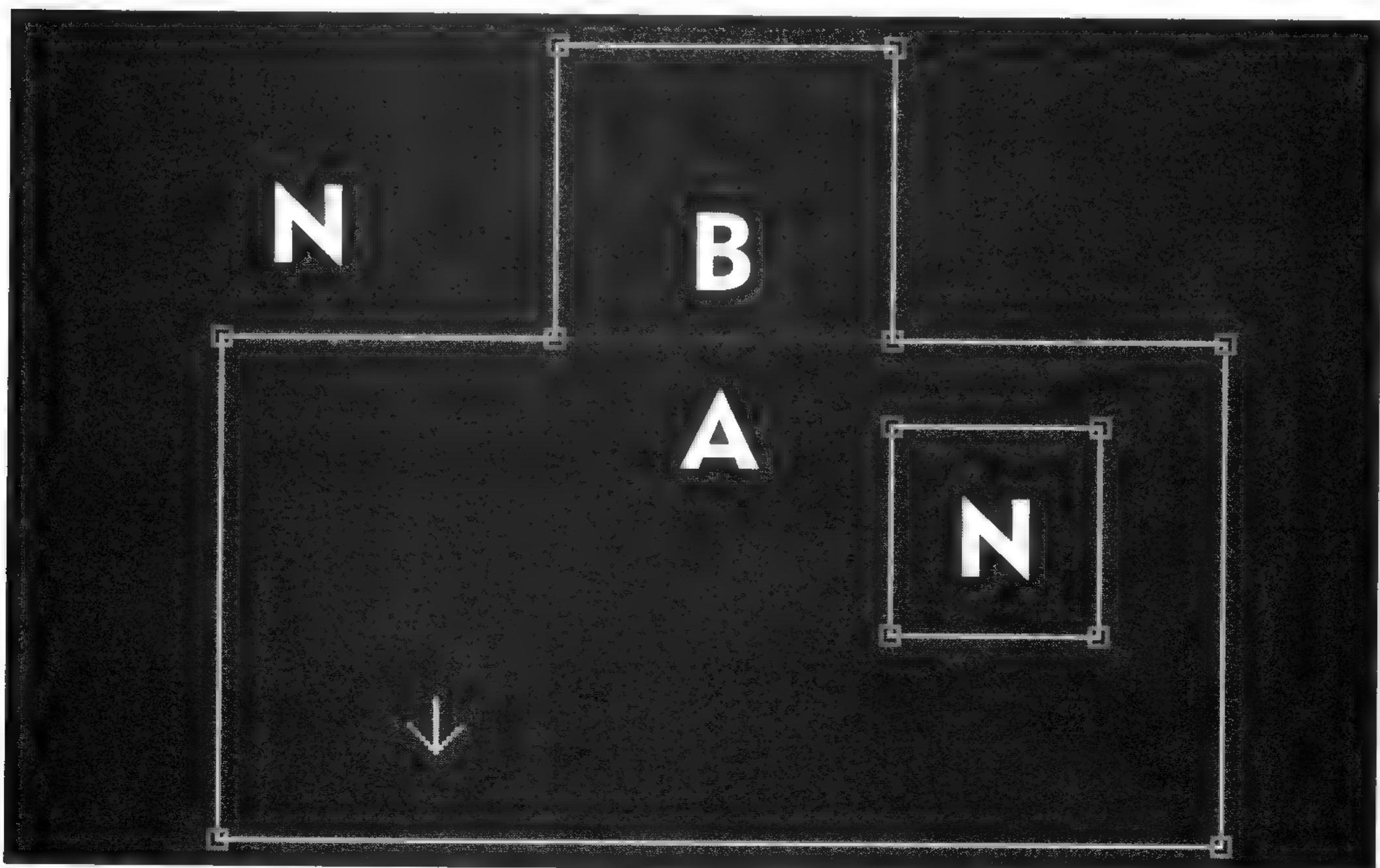


FIGURE 3.11: When you draw a rectangle inside an existing sector, you are actually carving out a region of null space inside the existing sector. Another step is required to make this new rectangle a child sector.

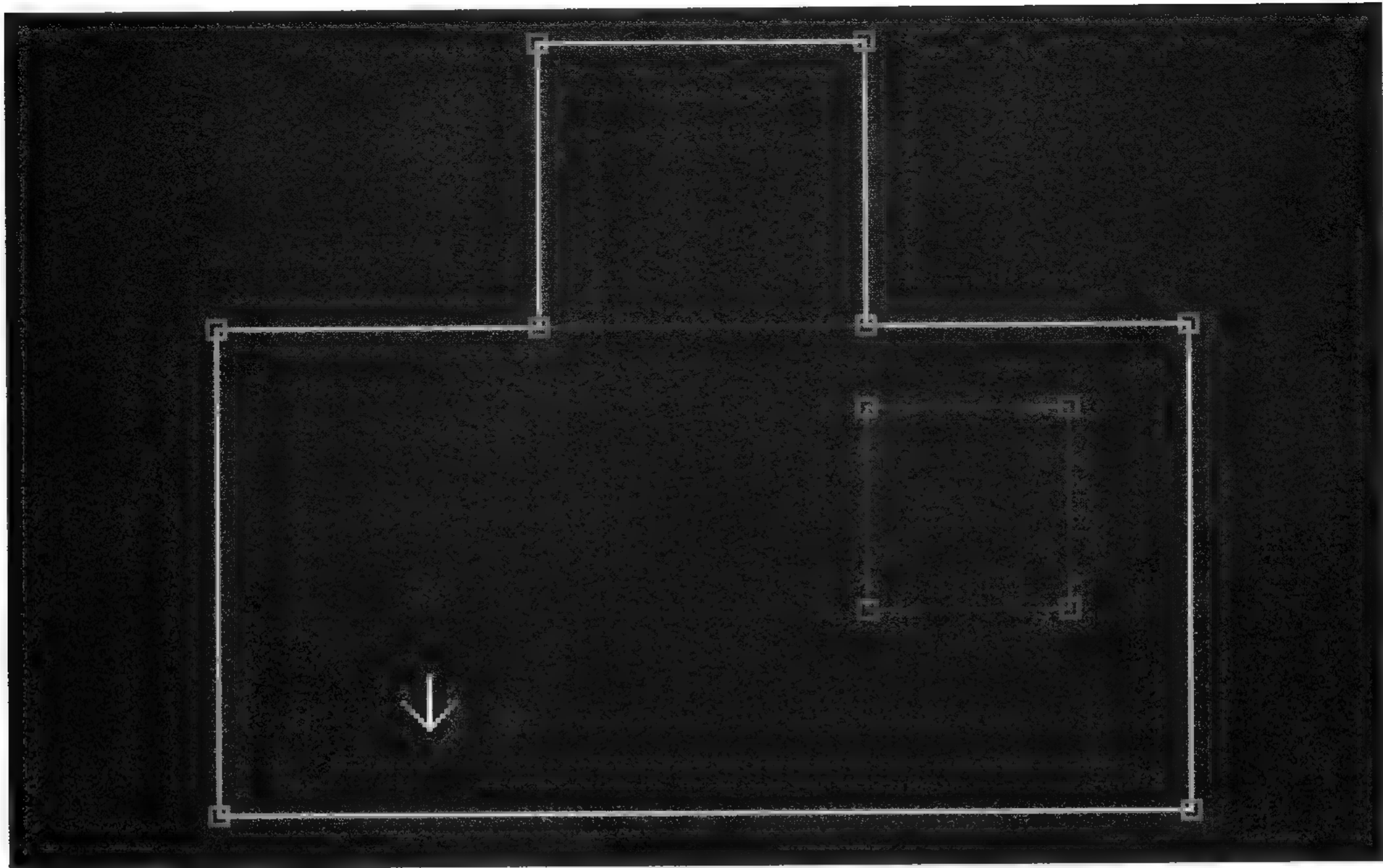


FIGURE 3.12: Notice in this completed child sector that its walls are now two-sided (red) instead of one-sided (white).

because its ceiling and floor have the same heights and textures as the parent sector. You can change the ceiling or floor heights the same way you did before, by using PgUp and PgDn. Remember to use the left mouse button to lock in the ceiling or floor once you start moving it. You also may have to fumble around a bit before you find the loca-

tion of the child sector, because it's impossible to see right now. If you do in fact place the mouse cursor on the parent sector by accident, the child sector's floor or ceiling height will *not* be altered when you press PgUp or PgDn; this should help you to find the child sector more easily.



QUICK FIXES

The original BUILDHLP documentation states that the key sequence to make a child sector is Alt+Ins. This is incorrect. The correct key sequence is Alt+S, as used in the example above.

SELECTING TEXTURES

Although your level is coming along, its walls, floors, and ceilings are still that same boring brown brick texture! It's time now to learn how to change the textures of the structures you create. In addition to making your sectors look more interesting, using different textures also helps you to differentiate your sectors in 3D mode.

You can choose textures in 3D view mode only. Place the mouse cursor on the wall, floor, or ceiling that you want to change. Then, press the V key. A list of all the textures used in the level so far appears that also tells you how many times each has been used, indicated by a little number in the upper-left corner of each texture. Notice that the brown brick texture is the only one showing because it's the only texture you've used up to now. To see *all* the textures available, press the V key a second time. This brings up all the available textures in numerical order, as you can see in Figure 3.13.

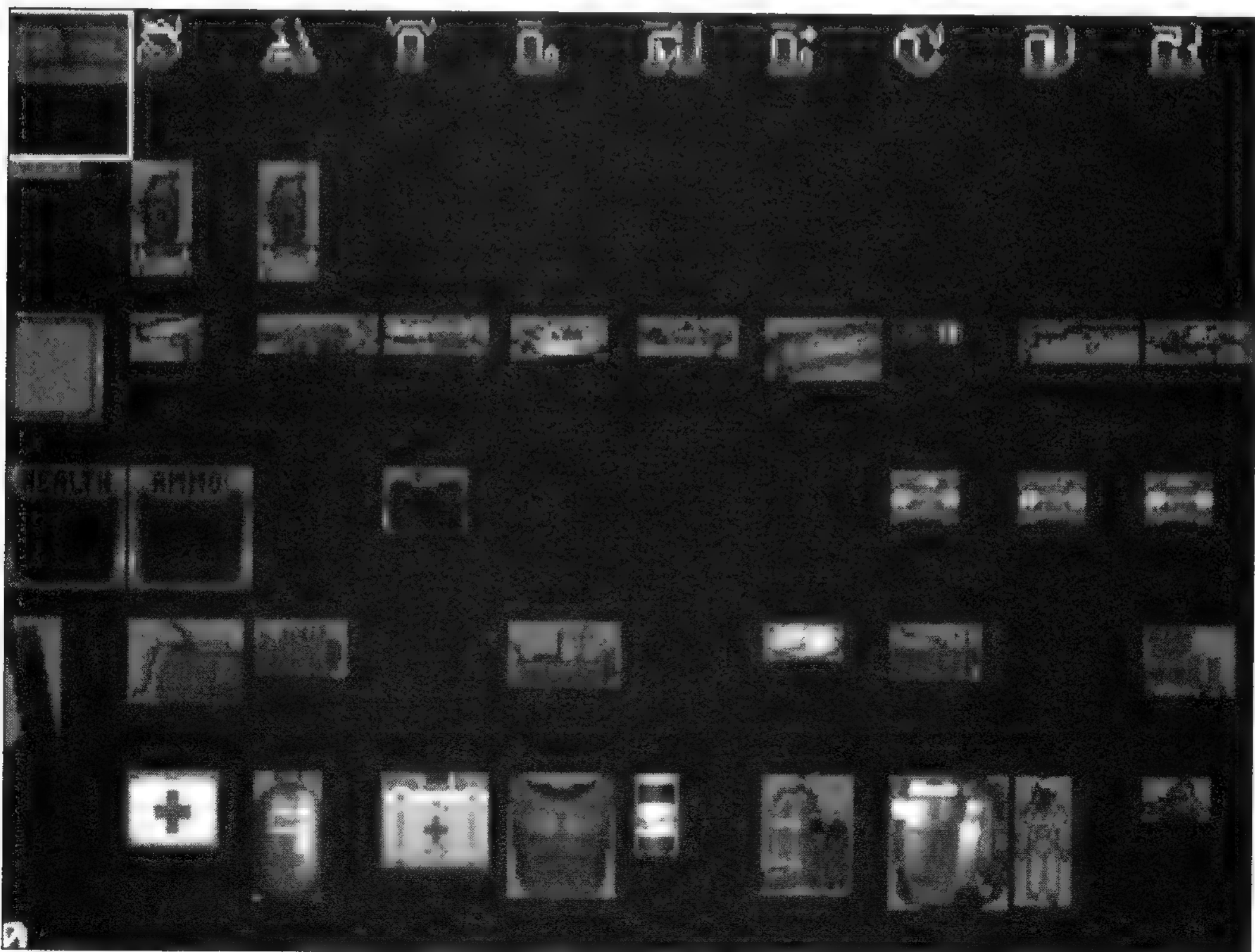


FIGURE 3.13: The texture selection screen displays all available textures you can apply to a wall.

You can use the arrow keys and PgUp and PgDn keys to scroll through this list. Or, if you know the number of the texture you're looking for, you can press G (for go to) and type that number. This jumps you right to it. Once you've found a texture you want to try, just press the Enter key, and voila! The texture gets painted right on the wall.

Very often you will paint the same texture on several connected walls, so Build has an easy texture copy and paste feature. To copy a texture, place the cursor on it in 3D mode and press Tab. Place the cursor on another wall and press Enter. The texture will be pasted onto that wall. You can easily spin around and paint all the walls of a room this way. Or, if you want to paint an entire sector *very* quickly, place the mouse cursor on a second wall in the sector and press Ctrl + Enter. The entire sector will be painted with the texture you copied using Tab.

Figure 3.14 shows an example of the same level shown in Figure 3.7 with some textures chosen for the three sectors. Different textures sure make a *huge* difference, don't they? (As Duke would say, "Ahhh, *much* better!")

There are many other functions for texture manipulation, and these will be covered in the next chapter. For now, you at least know how to change a texture so your eyes won't go buggy staring at that same ugly brown brick texture. Feel free to change wall, floor, and ceiling textures of the sectors you will create as you see fit.

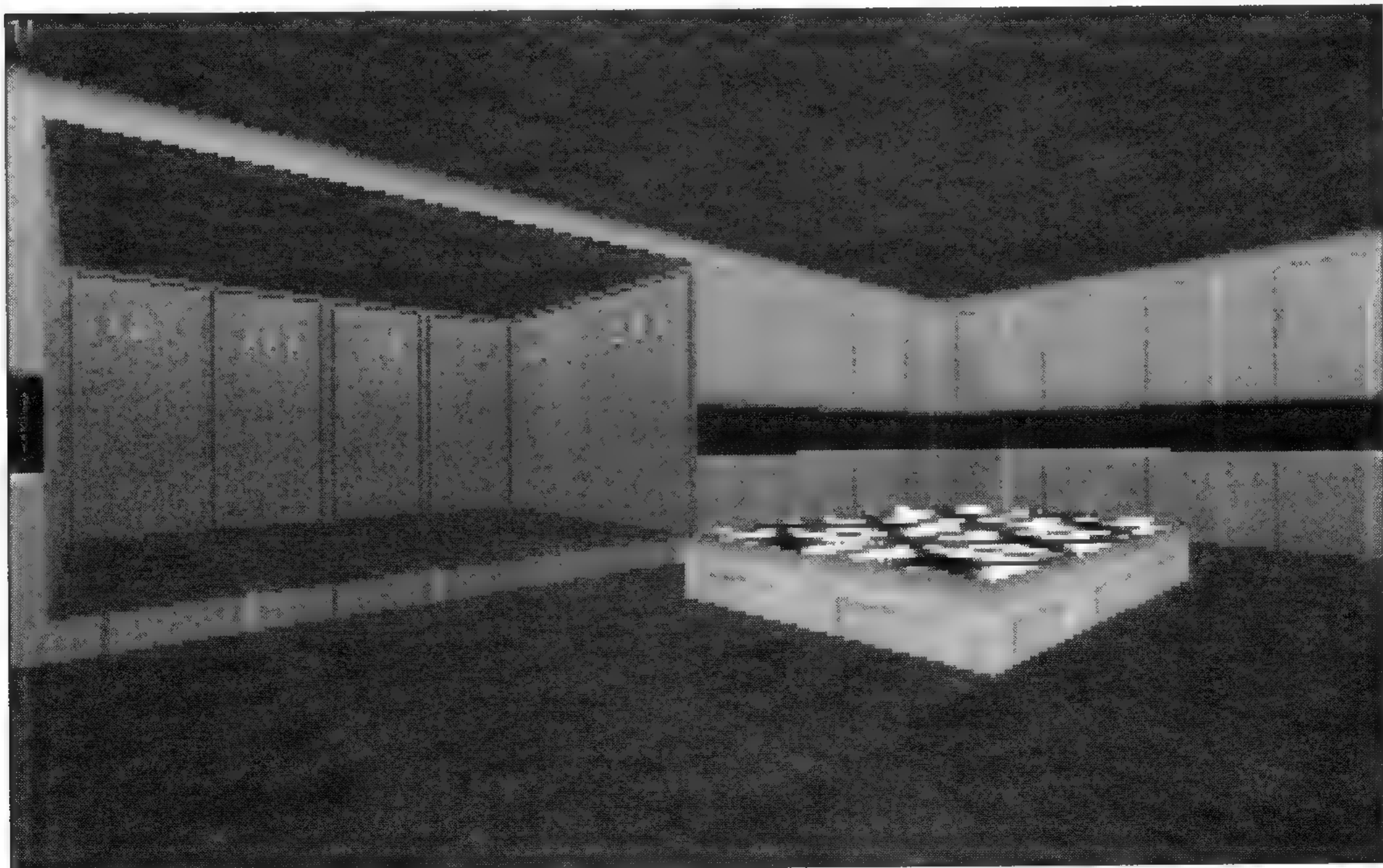


FIGURE 3.14: Choosing different textures makes all the difference in the world.

CREATING OTHER SECTOR TYPES

Apart from the child sector, which lies wholly inside a parent, there are two other types of sectors that lie partially within a larger sector. These are called the *peninsula* sector and the *corner* sector. Both are created very similarly, and an example of each is shown in Figure 3.15. One way to think of the shape of the parent sector, like the one shown in the figure, is to say that it has *bites* taken out of it. These bites are now occupied by new sectors. Let's take a look at creating the peninsula type first.

DRAWING THE PENINSULA SECTOR

The peninsula sector is usually square or rectangular in shape, and it has three walls lying within a parent sector. The fourth wall, however, lies on the same line as one of the parent sector's walls. This type of sector is useful for cabinet or kitchen-counter-type structures that are attached to a wall but have a raised floor to create the countertop.

To create a peninsula sector, start an entirely new room in another spot of your level map. The best way to get to a new spot is to zoom way out using the Z key while

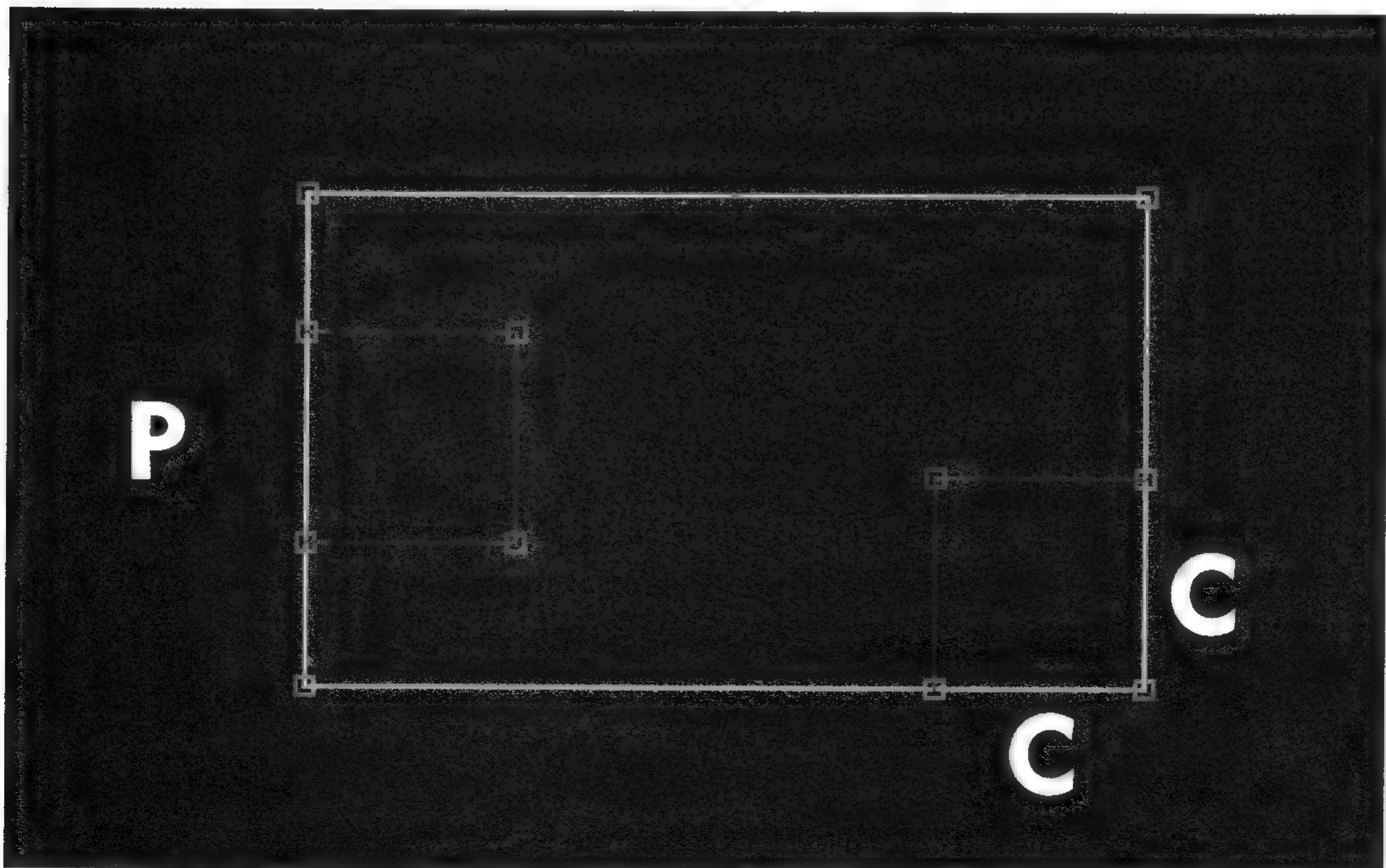
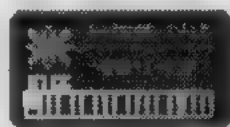


FIGURE 3.15: This parent sector contains a peninsula sector (west side) and a corner sector (east side).



TIP

You should have the key sequence for adding new vertices to a wall memorized by now, but just in case you need a reminder, it's the **Ins** key.



NOTE

Because your peninsula sector was actually created by splitting the original sector, you didn't have to complete the new sector by drawing the final wall (along the original west wall). This western wall is still one-sided because it defines **only** the peninsula sector; it doesn't define the parent sector in any way.

you are viewing the map in 2D view mode. Place the mouse cursor in a new spot, click the right mouse button to move the white arrow to that spot, and then zoom back in using the **A** key. This will quickly move you to a new place on the map.

To begin in this new area of the map, start by drawing a rectangular sector similar to the first sector you drew on the first part of the map. Your goal will be to create a peninsula sector along the west wall, as you see in Figure 3.15. Once the new room sector is drawn, start the peninsula sector by adding two new vertices to the west wall. Move the vertices if necessary to space them equally along the west wall.

Next, place the cursor on the northern of these two new vertices and press the spacebar to start the sector drawing process. Draw a line due east of this vertex (a few gridlines into the parent sector), and press the spacebar again. You should have a new line drawn from the west edge of the original sector running inward. Then continue drawing the sector by placing another vertex due south of the one you just

created. Finally, move the mouse to the southernmost vertex you inserted on the west wall (due west of the vertex you just created), and press the spacebar again. You should see the message "Sector split." on the status bar.

DRAWING THE CORNER SECTOR

The corner sector is created in a very similar way to the peninsula sector. To start a corner sector, two vertices are again added to the original sector, but this time they are added to different walls. You will add yours to the east and south walls of your level map. Place the cursor on the new east wall vertex, and press the spacebar to start the sector drawing process. Draw a new vertex due west of the starting vertex, and then end the process by clicking on the new south wall vertex. Once again, you should see

the “Sector split.” message. This time, you had to draw only two new walls of the corner sector; the two outer walls now become part of the corner sector. (See Figure 3.15.)

MOVING A SECTOR

You’ve seen that sectors can be moved by dragging individual vertices to other locations. This method can be cumbersome, however, especially if you need to move a large group of interconnected sectors over a few grid lines and you have to do it one vertex at a time. In addition to being a pain in the butt, it can be very difficult to retain the original shape of a sector you move, because moving one vertex immediately changes the shape of the sector.

Thankfully, there is a way to select a group of vertices and move them all at once. This means that you can take an entire completed section of your map and move it to a new area. You can also move a complete section over one grid line so that it connects with another area.

To select a group of vertices, start by viewing the map in 2D view mode. Position the mouse cursor in a corner where you will begin a rectangle that will enclose the group of vertices you want to select. Then, hold down the Shift key that’s to the *right* of the spacebar (referred to as Right Shift) and move the mouse cursor. A purple rectangle will be drawn from the spot where you press the Right Shift key to the new mouse cursor location. When you have a group of vertices within the rectangle, let go of the Right Shift key and all of these vertices will be selected as a group. You will be able to tell which vertices are selected because they will be flashing. You can now use the mouse to drag the selected vertices to another map location at the same time. When you have the vertices in their desired location, press the Right Shift key once again, and the selected vertices will no longer be selected.

DELETING A SECTOR

If you end up creating a sector that you no longer want, you can delete an entire sector all at once. All walls that define this sector will be deleted automatically. If a two-sided wall helps to define a sector that you delete, that wall will become one-sided and

**SEE**

Sprites, objects that you can place within a sector's design, will be covered in chapter 5.

**TIP**

Be very sure before you delete a sector because there is no Undo feature in Build, and the only ways to bring it back are to either recreate it or reload the file from disk. For this reason, it may be a good idea to save the level just prior to deleting any sectors.

will define only the sector that was attached to the deleted sector via this wall.

To delete a sector, just place the mouse cursor inside it while in 2D mode and press Right Ctrl + Del (the Ctrl key to the *right* of the spacebar). All one-sided walls that make up this sector will be deleted, and two-sided (red) walls will be turned into one-sided (white) walls. Orphaned vertices will also be deleted, as well as any sprites (objects) currently in that sector.

Figure 3.16 shows an example of what occurs when existing sectors are deleted from a map. In this case the peninsula and corner sectors were deleted from the example sector shown in Figure 3.15.

Notice in Figure 3.16 that the walls (two-sided) that were red previously are now white

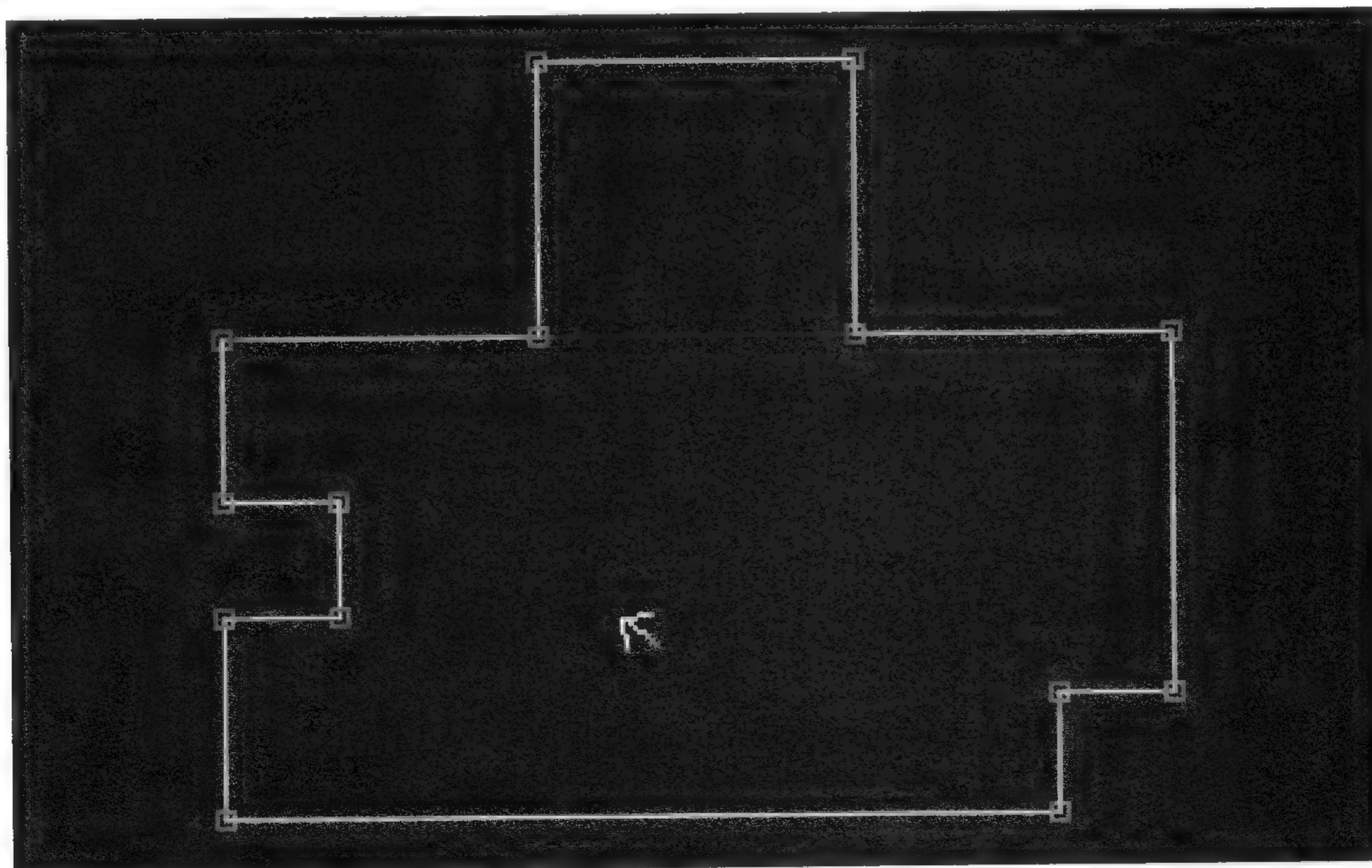
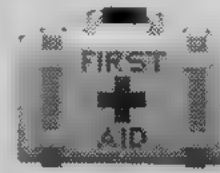


FIGURE 3.16: The peninsula and corner sectors shown in Figure 3.15 have been deleted in this example.

**WARNING**

Don't press Ctrl+Alt+Delete by accident (as I've done once or twice), instead of Ctrl+Del. You'll of course reboot the PC. This is yet another good reason to save the level before deleting a sector.

**TIP**

You can copy more than one adjacent sector at the same time by enclosing all the sectors you want to copy within the green rectangle.

because they are now one-sided. Also, the walls exclusive to the deleted sectors are gone completely.

COPYING A SECTOR

Copying a sector is an important function to learn because many of the special effects covered in later chapters (like creating underwater areas) require creating two sectors of the *exact* same dimensions. In this case it's often easiest to make one sector and copy it. Copying a sector can be pretty tricky, so pay close attention in this area.

To begin copying a sector while in 2D mode, put the mouse cursor to the north and west of the sector you want to copy. Hold down the Alt key that's to the *right* of the spacebar (referred to as Right Alt). While holding down Right Alt, drag the mouse cursor to the right and below the sector. You'll see a green rectangle expand from the point where you pressed Right Alt to where you move the mouse cursor. Make the rectangle completely surround the sector you want to copy, and then release the Right Alt key. The sector that was enclosed in the green rectangle will now be filled with green flashing horizontal lines, as you can see in Figure 3.17. If you don't see the green lines on your screen, try selecting the sector again.

Once you have the sector selected, place the mouse cursor inside the sector, hold down the left mouse button, and press Ins. Now, while *still* holding down the left mouse button, move the mouse cursor and a copy of the selected (green) sector will follow. Bring the copied sector to the place you'd like to put it and *then* release the mouse button. This will drop the sector into place, as shown in Figure 3.18. Finally,

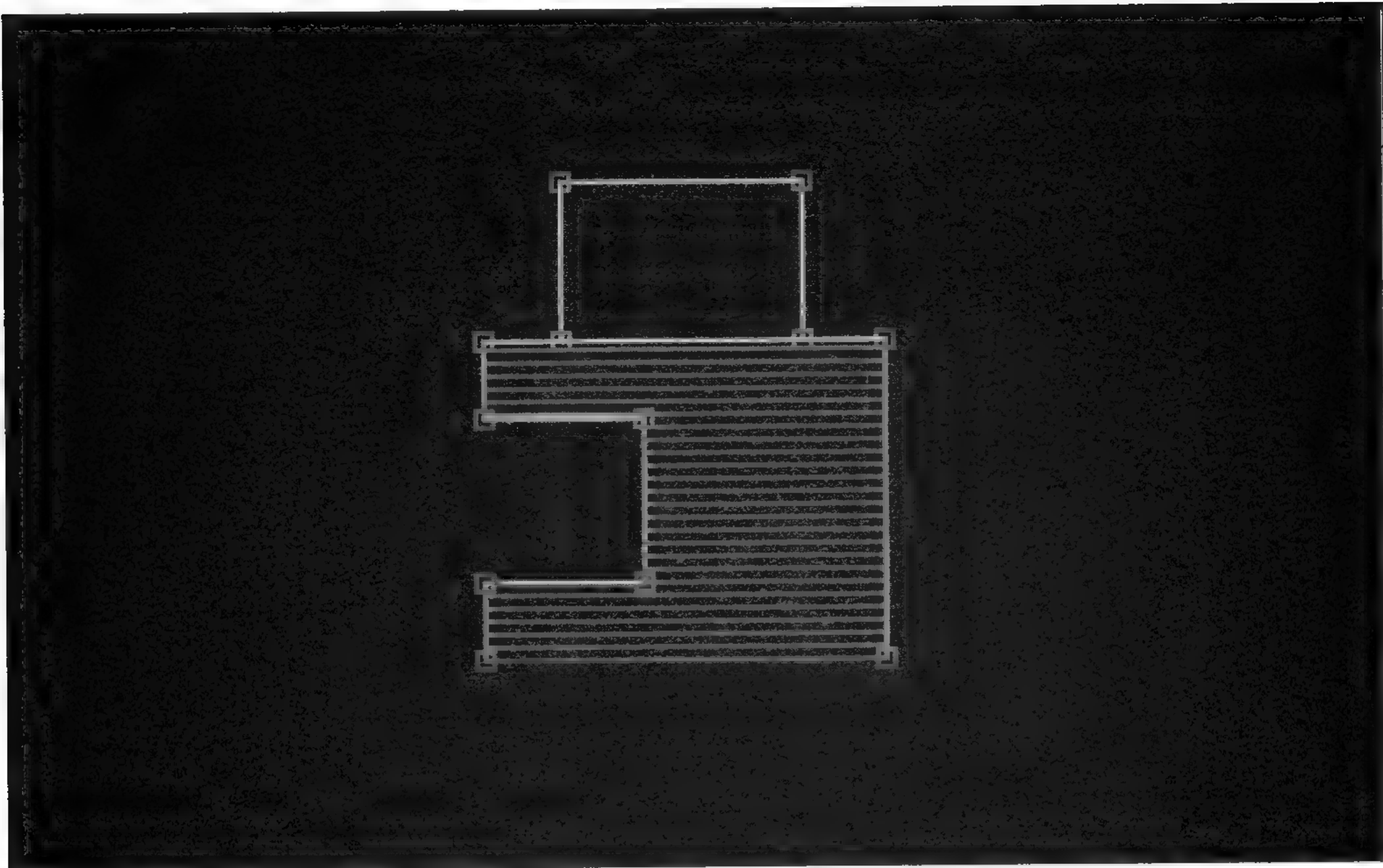


FIGURE 3.17: When a sector is selected for copying, it will be filled with green flashing horizontal lines.

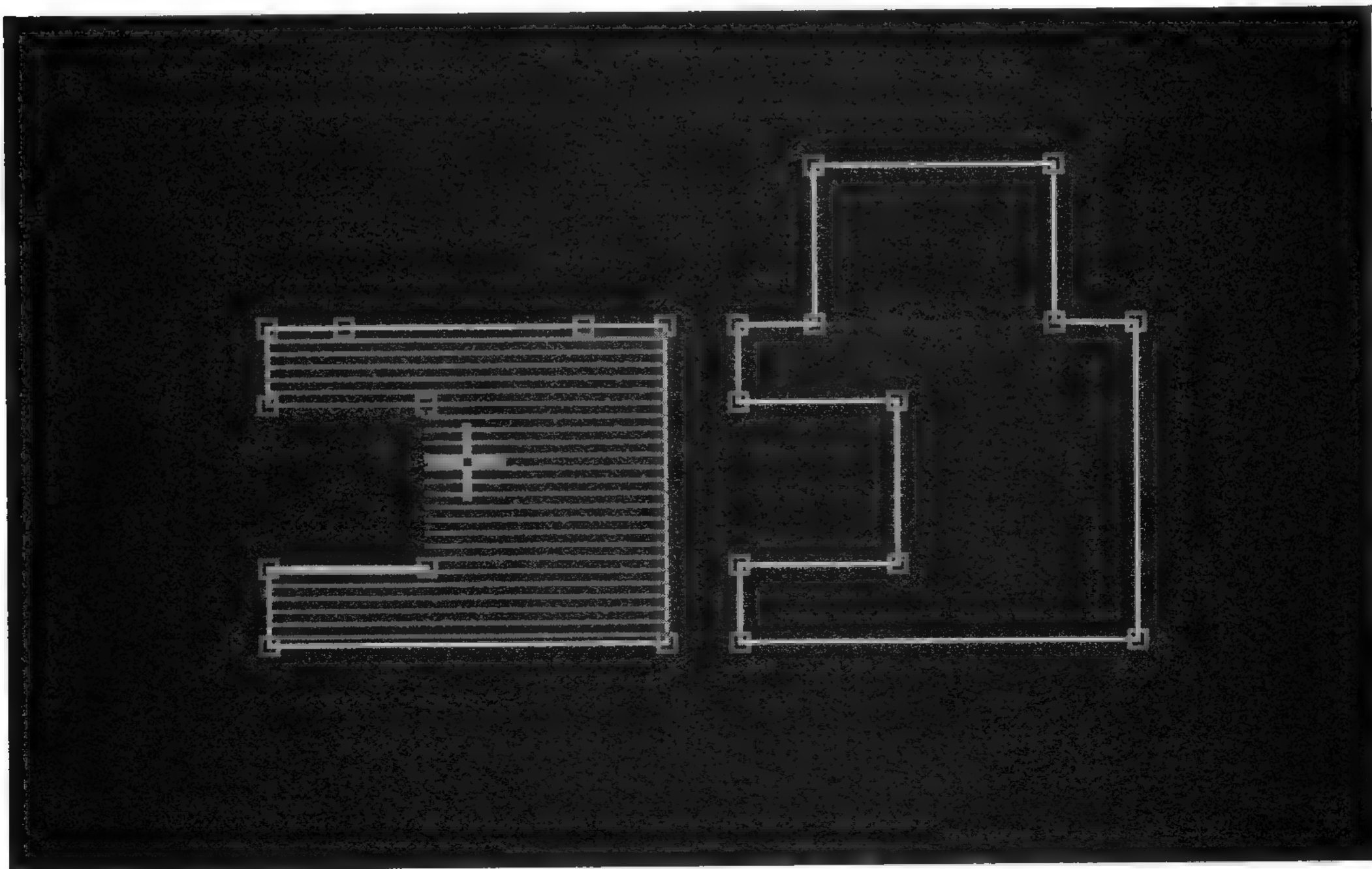


FIGURE 3.18: After you place a selected sector copy in its location, the copy is the green-highlighted sector and the original sector remains in its original location.

press the Right Alt key again to deselect the green sector.

COPYING CHILD SECTORS

The steps for copying sectors discussed previously work fine if the sectors you wish to copy are not *child* sectors. When you want to copy *child* sectors, some extra steps are required. You will practice this operation by first creating a parent sector with a child sector inside and then creating another sector with a *template* sector inside.

After you create the small, five-sided child sector in the left room as shown in Figure 3.19, your goal will be to make a copy of it and move the copy into the right room. First create the right room, and then make a smaller *template* sector inside it that lies just within its borders (see Figure 3.20). Do **not** make this sector a child sector by pressing Alt + S; instead, leave the walls white for now.



NOTE

If you accidentally release the mouse button before you can move the copied sectors, they will drop right in place over the original sectors, and you will have two sets of sectors occupying the same place on the map. To avoid this rather undesirable structure, you should always save your map before copying a sector or sectors.

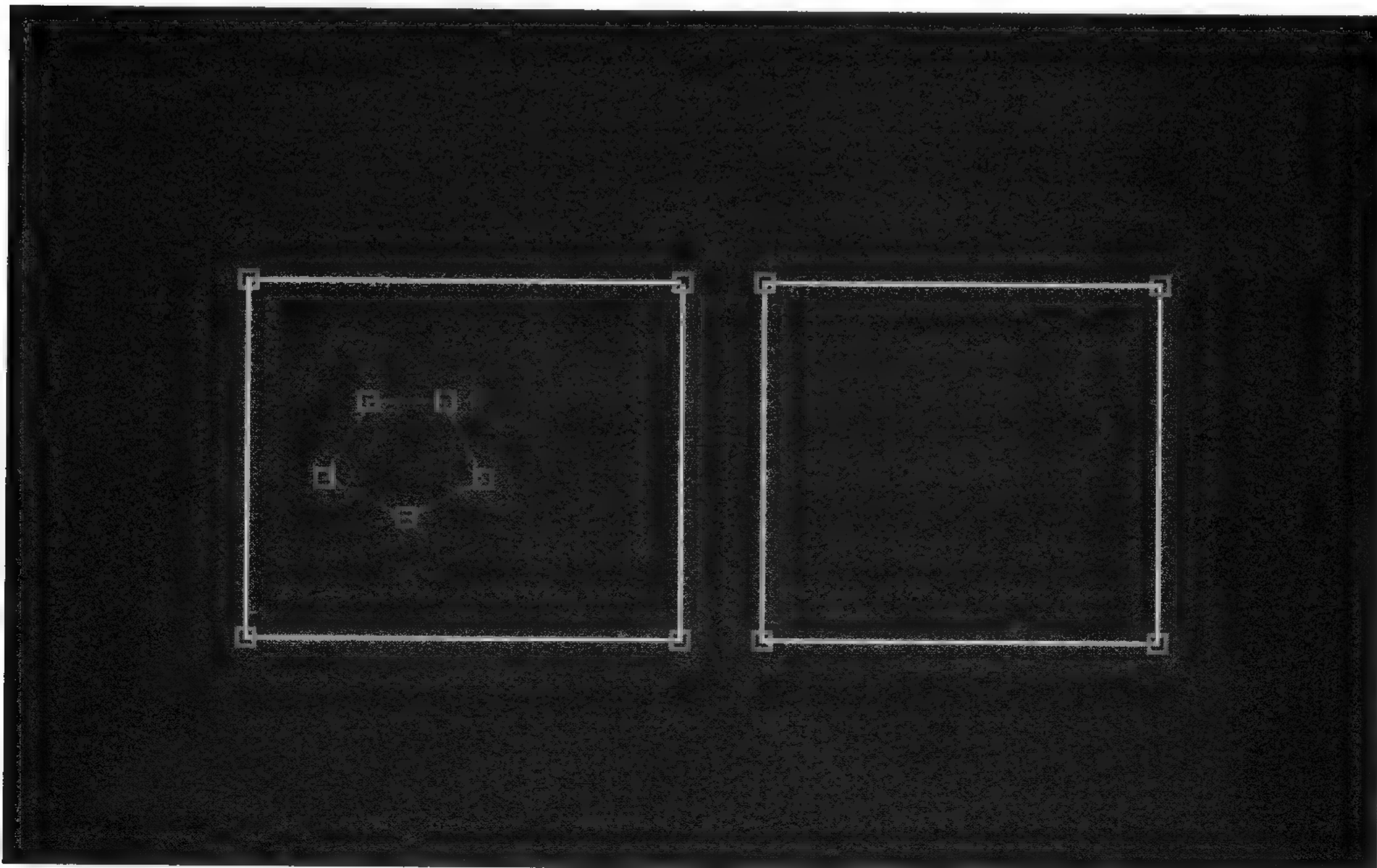


FIGURE 3.19: Set up an area of your map for copying the pentagonal child sector from the west room to the east room.

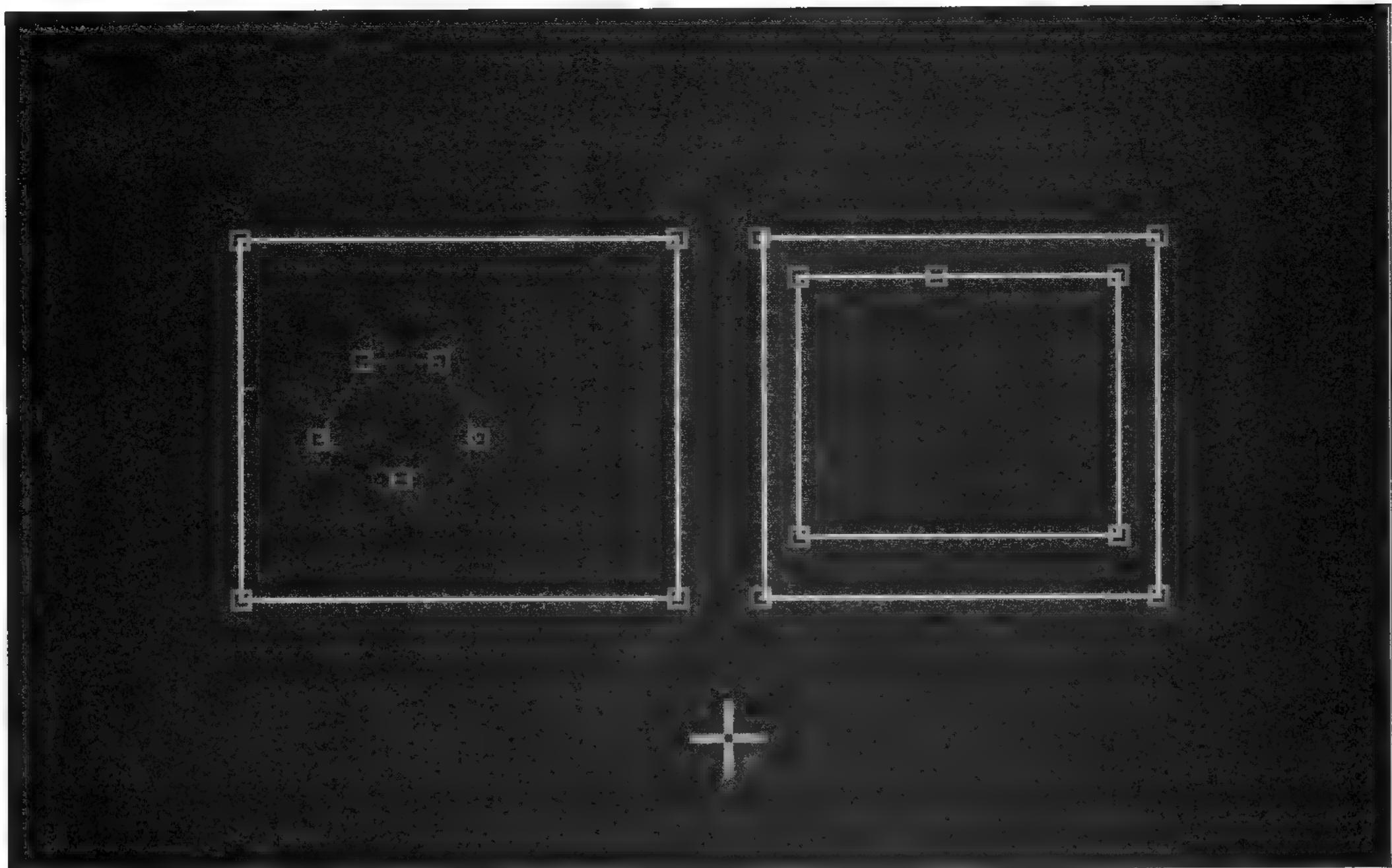


FIGURE 3.20: Make a template sector similar to the one on the right that will receive the copied sector. Do not use Alt+S to turn this sector into a child sector.

Next, select the child sector in the left parent sector for copying by using the Right Alt and Ins keys as described in the previous section. Use the mouse to move the copied sector so it's inside the template sector you just made on the right side.

The final step is to move each of the vertices of the template sector and drop them right on a corresponding sector on the copied child sector. Because the copied child sector has five vertices and the template sector may have only four, be sure to split one of the template sector's walls first to give it five vertices. Once you've done this, drag each of the template sector's vertices in turn and drop them on one of the corresponding vertices of the copied child sector. Once you do this, each of the copied child sector's walls should turn red.

Note that right now this method of copying child sectors seems like more work than recreating the sector manually. However, when you copy a sector, all of the sprites (objects) and special sector definitions within that sector are also copied. If you have several objects placed in your source sector that you will also need to copy, then this method will be faster than creating the sector *and* every object again.

One final point about copying sectors: You can also copy them out of one MAP file and into another. To do this, repeat the steps discussed previously for selecting and

copying a sector. While the copy is still blinking green, press Esc and then load the level that is to receive the sector copy. The new sector (copy) will appear in the exact same place as on the original level map. Because a sector copies to the exact same coordinates, you may have to zoom the source map way out to find the newly placed copy. You'll know it once you find it because it will still be blinking green, which means you can still drag it to another location. When the copy is in the right spot, press Right Alt one final time to remove the green selection lines.

ROTATING A SECTOR

When you select one or more sectors with the Right Alt key, pressing the , (comma) key or the . (period) key will rotate a selected (blinking green) sector by 90 degrees. You can hold down Shift while using these keys to rotate the sector in much smaller increments. The , (comma) key rotates a sector clockwise; the . (period) key rotates a sector counterclockwise. Using the rotating operation in combination with the copying operation is very useful for creating sectors that are mirror images of each other. Figure 3.21 shows an example of a selected sector copy that is rotated 90 degrees.

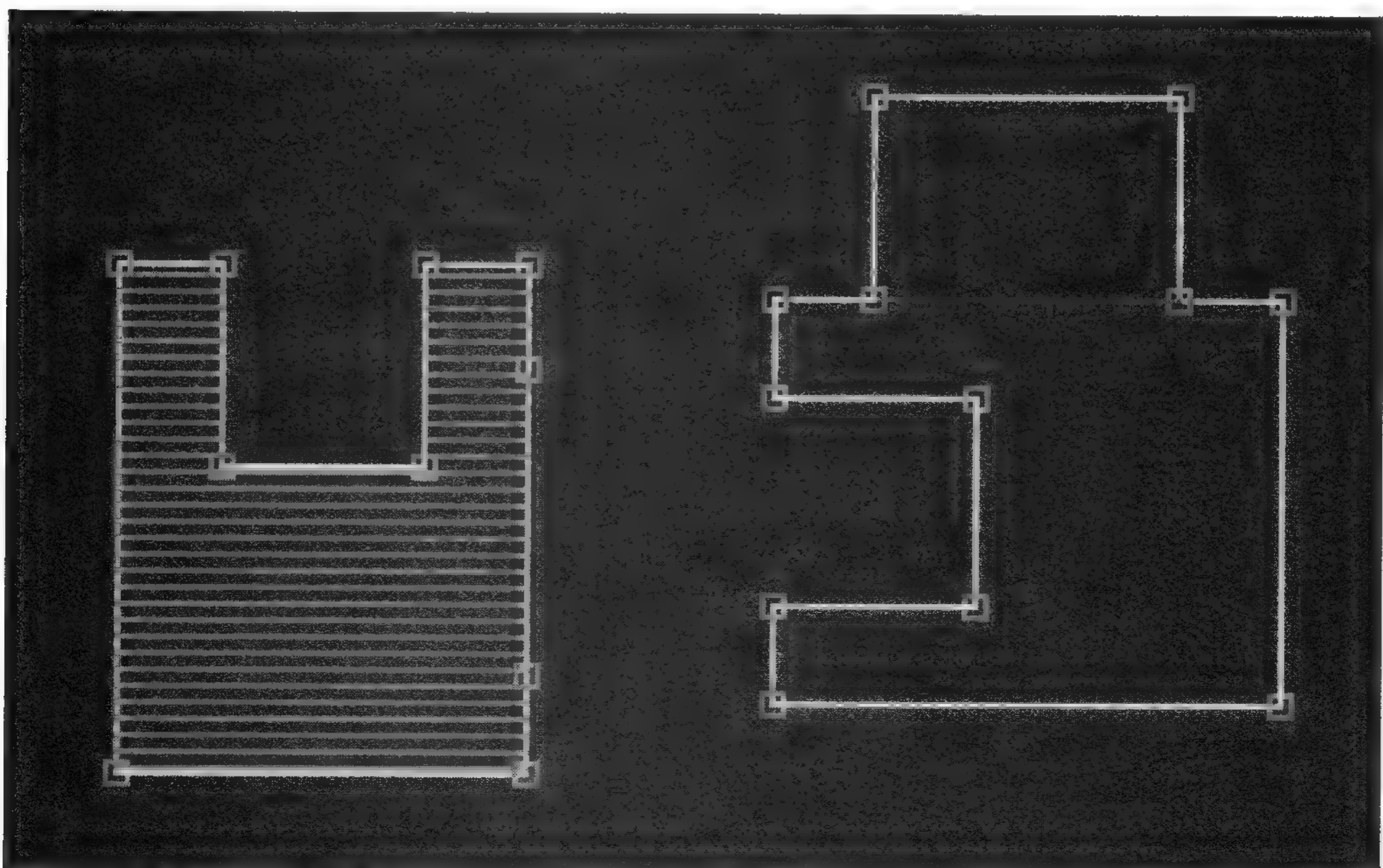


FIGURE 3.21: Here is a selected sector copy that is rotated 90 degrees.



SLOPING FLOORS AND CEILINGS

The Build engine is capable of drawing sloped floors and ceilings, which can be used for structures like ramps or peaked roofs. Sloping a floor or ceiling is done in 3D mode. Go into 3D mode, place the mouse cursor on the floor or ceiling that you wish to slope, hold down the left mouse button to lock that surface, and use the [or] key. The [key slopes the floor upward from the anchor wall, and the] key slopes the floor downward. You can also slope the surface in much smaller increments by holding down Shift while you use the [or] key. You can also automatically align the slope of a floor or ceiling with that of an adjacent floor or ceiling by using Alt + [and Alt +].

CONTROLLING THE DIRECTION OF SLOPE

When a floor or ceiling slopes, one of the walls remains in the same place as the *anchor wall*, and the slope occurs in a direction perpendicular to that wall. The name for this anchor wall is the *first wall* because Build uses the first wall that you draw when you create a sector as the wall that controls the direction of slope. It would be very difficult

to draw all of your sectors and keep in mind which wall you wanted to use as the anchor wall, however, so Build gives you a way to set whichever wall you want as the *first wall*. To set the first wall in 3D mode, just put the mouse cursor near that wall and press Alt + F. Once you set the first wall of a sector, the floor or ceiling will slope in a direction perpendicular to the direction of this wall.

Figure 3.22 shows an example of a sloped ceiling from an actual game level map. Notice that this room is actually made of several sectors (including the crates and the skylight), so several sectors' ceilings had to be sloped to the same angle and aligned perfectly to achieve the effect of a single, solid ceiling.



TIP

You can also set the first wall in 2D mode using Alt+F. Setting the sector's first wall will also be important when you work with textures in chapter 4.



FIGURE 3.22: Here's a great example of a sloped ceiling at the end of the Red Light District level in L.A. Meltdown (E1L2).

CREATING SECTORS OVER SECTORS

The Build engine is one of the first 3D game engines that can support sectors that lie over other sectors. This was the primary limitation of the Doom engine: For every x-y coordinate on a map, there was only one floor and one ceiling (i.e., one sector). *Duke Nukem 3D* allows sectors over sectors, but there are still some limitations. The main limitation is that you cannot design one sector over another and have both visible at the same time. In other words, creating a three-story building with windows on all sides won't work because the player would be able to see both stories at the same time through the windows. Catwalks are also out for the same reason; the player could see both sectors at the same time.

Now, I know what you're thinking: "What about the catwalk at the end of the very first level of the game?" Well, I sort of lied before, you *can* create catwalks, but it involves using a special kind of *sprite* (an object) for the bridge. See chapter 6 for a discussion of special sprites, including those for creating bridges.

To create an example of a sector over another sector, start by making a rectangular sector with the longer walls on the north and south sides. Raise both the ceiling and the floor of this sector about 20 clicks, so that it will be physically over the second sector you will create. Now, make another rectangular sector, this time with the longer walls on the east and west sides, directly over the first sector, as shown in Figure 3.23. (Don't worry about the white lines crossing.) The one thing that's critical to remember is to *not* try to make any of the vertices from the upper and lower sectors in the same place; Build will try to join these two sectors if you place one vertex on top of the other. You can then connect these sectors using such structures as an elevator or stairs. (Stairs are merely small sectors with different floor heights; elevators are special sectors that will be covered in chapter 7.) Also note that the main drawing limitation in the Build engine rendering is that Build cannot draw two sector floors or ceilings that lie on top of each other in the same view. So any stairs or holes you make must not allow the player to see both ceilings or both floors of a stacked sector at the same time.

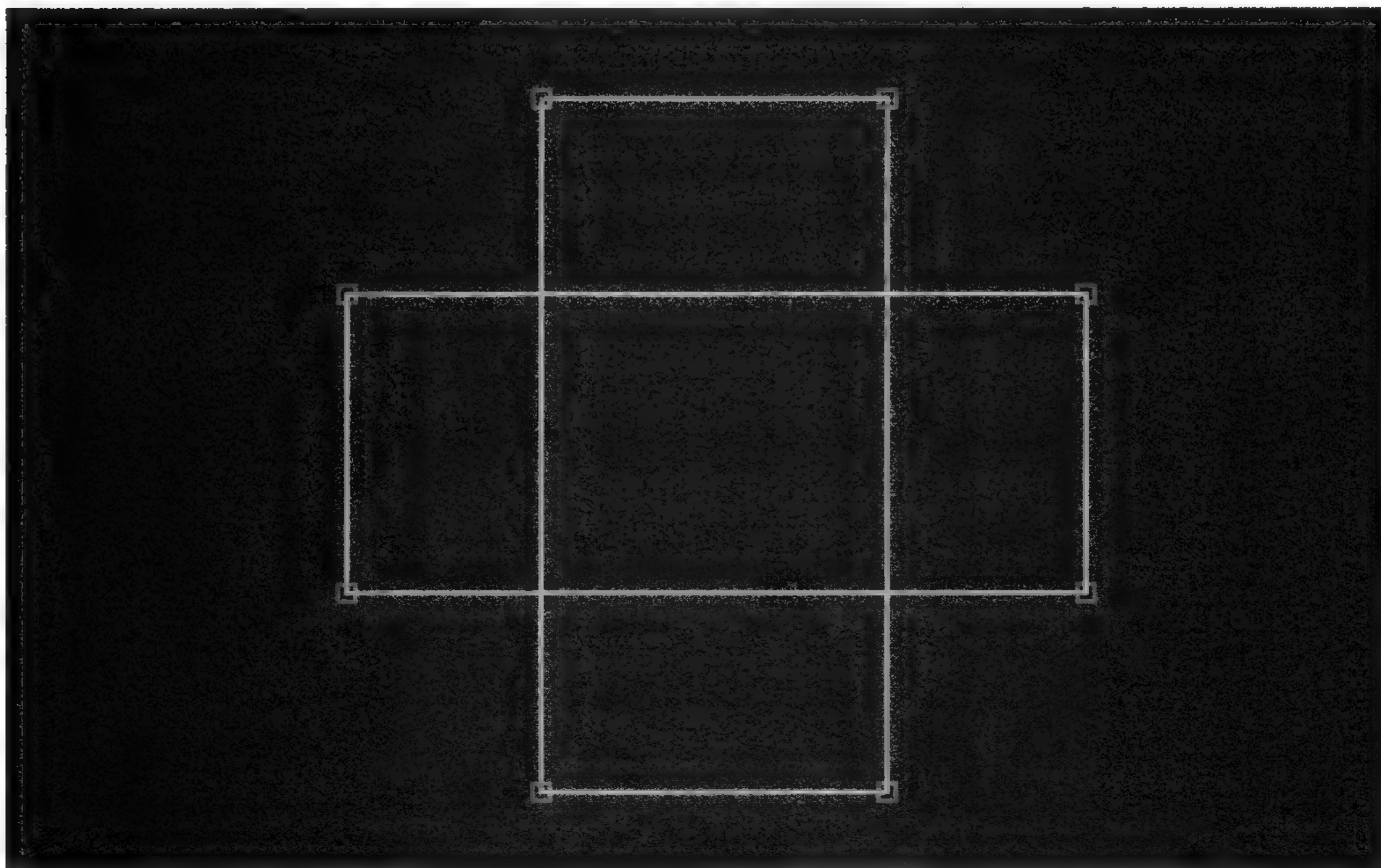


FIGURE 3.23: Creating one sector over another requires creating separate sectors with different floor and ceiling heights so they don't occupy the same space.

DO SOME ARCHITECTURAL INVESTIGATION

Believe it or not, you now know the basics of creating every type of structure in *Duke Nukem 3D* levels. Your next task, should you choose to accept it, is to *study* and to *practice*. Go back and play some of the levels in the game with which you're most familiar. Take time to wander around the level, looking at the structures. Try to figure out how each one was made. (You may want to enable God mode so you won't be disturbed.) Load the level into Build, and see if you were right. Ignore the doors, elevators, teleporters, and moving objects for now; we'll get to those objects later. Open up an empty level and see if you can replicate some of those structures. When you think you can create all the basic structures you've seen, it's time to go on to chapter 4.

KEYSTROKE AND OPERATION SUMMARY

A summary of all of the keystroke sequences and operations covered in this chapter appears in Table 3.1.

| TABLE 3.1: CHAPTER 3'S KEYSTROKES AND OPERATIONS | | |
|---|-------------------------------|-----------------|
| KEY SEQUENCE/OPERATION | FUNCTION | VALID VIEW MODE |
| A | Zoom in | 2D |
| Z | Zoom out | 2D |
| Left mouse button (standard drag) | Move a vertex | 2D |
| Ins | Split a wall (inserts vertex) | 2D |
| PgUp (with mouse on floor/ceiling and left mouse button held down) | Raise floor/ceiling height | 3D |
| PgDn (with mouse on floor/ceiling and left mouse button held down) | Lower ceiling height | 3D |
| Connecting 2 vertices of same sector during sector drawing mode | Split a sector | 2D |
| J (with mouse on first sector), J again (with mouse on second sector) | Join a sector | 2D |
| Drag vertex onto neighbor | Delete a vertex | 2D |
| Alt+S (with mouse inside sector to become the child sector) | Create a child sector | 2D |

(Continued on next page)

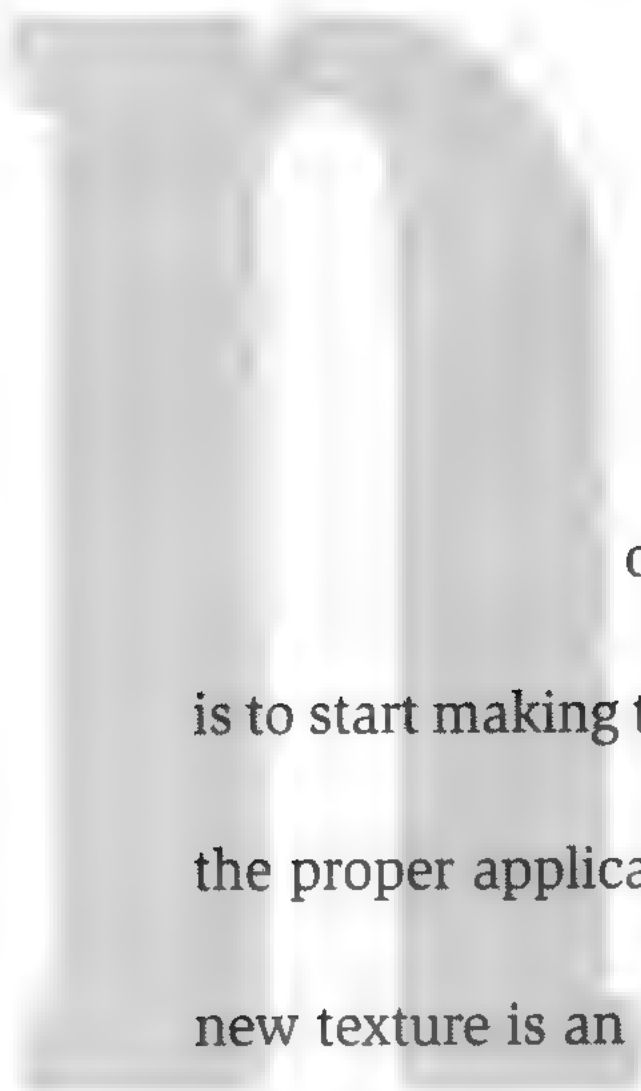
(Continued from previous page)

| KEY SEQUENCE/OPERATION | FUNCTION | VALID VIEW MODE |
|--|---|-----------------|
| V (once), Enter | Choose a texture from those used in current level | 3D |
| V (twice), Enter | Choose a texture from all available textures | 3D |
| Tab | Copy a texture to the clipboard | 3D |
| Enter | Paste a texture from the clipboard | 3D |
| Ctrl+Enter | Paste the texture in the clipboard on all walls in a loop | 3D |
| Right Shift (while moving mouse to enclose vertices in rectangle) | Select multiple vertices | 2D |
| Drag selection rectangle while holding Right Shift key | Move all vertices at once | 2D |
| Right Ctrl+Del | Delete a sector | 2D |
| Right Alt (while moving mouse to enclose sectors in a rectangle) | Select a sector | 2D |
| Ins (with left mouse button held down over a selected sector) | Copy a sector and drag the copy to a new location | 2D |
| , (comma) with one or more sectors selected | Rotate sector or sectors 90 degrees clockwise | 2D |
| . (period) with one or more sectors selected | Rotate sector or sectors 90 degrees counterclockwise | 2D |
| Shift+, (comma) | Rotate sector or sectors clockwise one degree at a time | 2D |
| Shift+. (period) | Rotate sector or sectors counterclockwise one degree at a time | 2D |
| [or] (with mouse cursor on floor or ceiling and left mouse button held down) | Slope a floor or ceiling | 3D |
| Alt+[or Alt+] | Align slope of a floor or ceiling with that of an adjacent floor or ceiling | 3D |
| Alt+F | Change sector's first wall | 2D or 3D |





Bringing Sectors to Life



Now that you can create sectors of all shapes and sizes, your next step is to start making them look like realistic architectural structures. The secret to this is the proper application of texture, shade, and color. Giving a wall, floor, or ceiling a new texture is an easy task. Giving it the *right* texture for the right situation is what you'll have to get good at if you want to make decent levels. Fortunately, the designers of the original *Duke Nukem 3D* levels were true masters of their craft. This means that you have *great* material to learn from. In this chapter you will continue to work with Build as you learn the mechanics of applying texture, light, shade, and color. Then the chapter will conclude with several case studies from the game's second episode, Lunar Apocalypse. This will give you a sense of not only *how* to apply textures, shade and color, but how to apply them *well*.

EMBELLISHING WALLS

You learned how to apply wall textures in 3D mode in the last chapter, if only for the purpose of keeping your sanity so you didn't have to look at the same drab brown brick texture on every surface. Here you will look at a variety of ways in which you can apply and change textures using a variety of techniques.

CHANGING WALL TEXTURES

To change a texture on a wall, move the mouse cursor on that wall and press the V key. The first screen that comes up is a list of all the textures used on the walls so far, sorted in order of the frequency that they've been used (a small white number in the upper right of each texture shows how many times it's been used).

If you want to look at *all* the available textures, press the V key a second time, and the texture list will come up in numerical order. Note that *all* of *Duke Nukem 3D*'s graphics appear in this second texture display, including textures normally used for objects or monsters. To select the texture that you want, use the arrow keys, PgUp, or PgDn, and a white rectangle (the texture selector) will move from texture to texture. When you find the texture you want to use, move the texture selector to that texture, and then press the Enter key. You will see the texture you chose mapped onto the wall.

There are over 3,400 textures to choose from, so applying textures is an acquired art, especially when it comes to building sectors that look realistic and enhance the overall game experience. Unfortunately, there's no substitute for practice to become familiar with all the textures and see which ones look good in which situations.



NOTE

Build's textures are grouped by special names that appear in all uppercase characters in this book (for example, MOON-SKY1). Textures are also identified by numbers. These are shown in this book with a pound sign (#) followed by the number (for example, texture #200).

Using Different Textures on Top and Bottom Wall Sections

Some transparent (two-sided) walls have two sections—for example, a wall bordering a window sector will have a solid surface below the window (a *step*) and a solid surface above the window (a *ledge*). (If you have any experience in *DOOM* level editing, these surfaces are known as the WallAbove and WallBelow textures.) Normally, when you apply a texture to this type of wall, both the top and bottom surfaces will be painted with the same texture. However, certain situations may require that you make these two sections a different texture.

To allow the top and bottom textures of a two-sided wall to be separately editable, place the mouse cursor on the upper part of the wall section and press the 2 key (*not* the numeric keypad's 2 key, the one above the W key on the keyboard). This will cause

the lower part of the wall to revert to the default brown brick texture. The lower texture can now be edited separately from the upper texture. Figure 4.1 shows a two-sided wall with different textures on its top and bottom sections.

Creating Masked Walls

In some situations, you may need to place a texture right on the surface of a two-sided wall. The best example of this is when you make a glass window. A glass window is created by forming a standard two-sided wall and then applying the glass texture (#503) to the surface of the wall. Another example would be applying a texture to a normally transparent wall to hide the sector beyond from view. This is a good method for creating secret areas.

A two-sided wall with a texture on its surface is called a *masked wall*. (See Figure 4.2.) To create a masked wall, switch to 3D view mode, place the mouse cursor on the lower section of the wall (or just below the wall if it has no lower section), and press the M key. You will see the transparent area of the wall fill with the default texture. You may now edit that texture normally. The texture that you put on a masked wall will be seen from both sides of the two-sided line. Sometimes, it may be desirable to



FIGURE 4.1: This two-sided wall has different textures on its top and bottom sections.

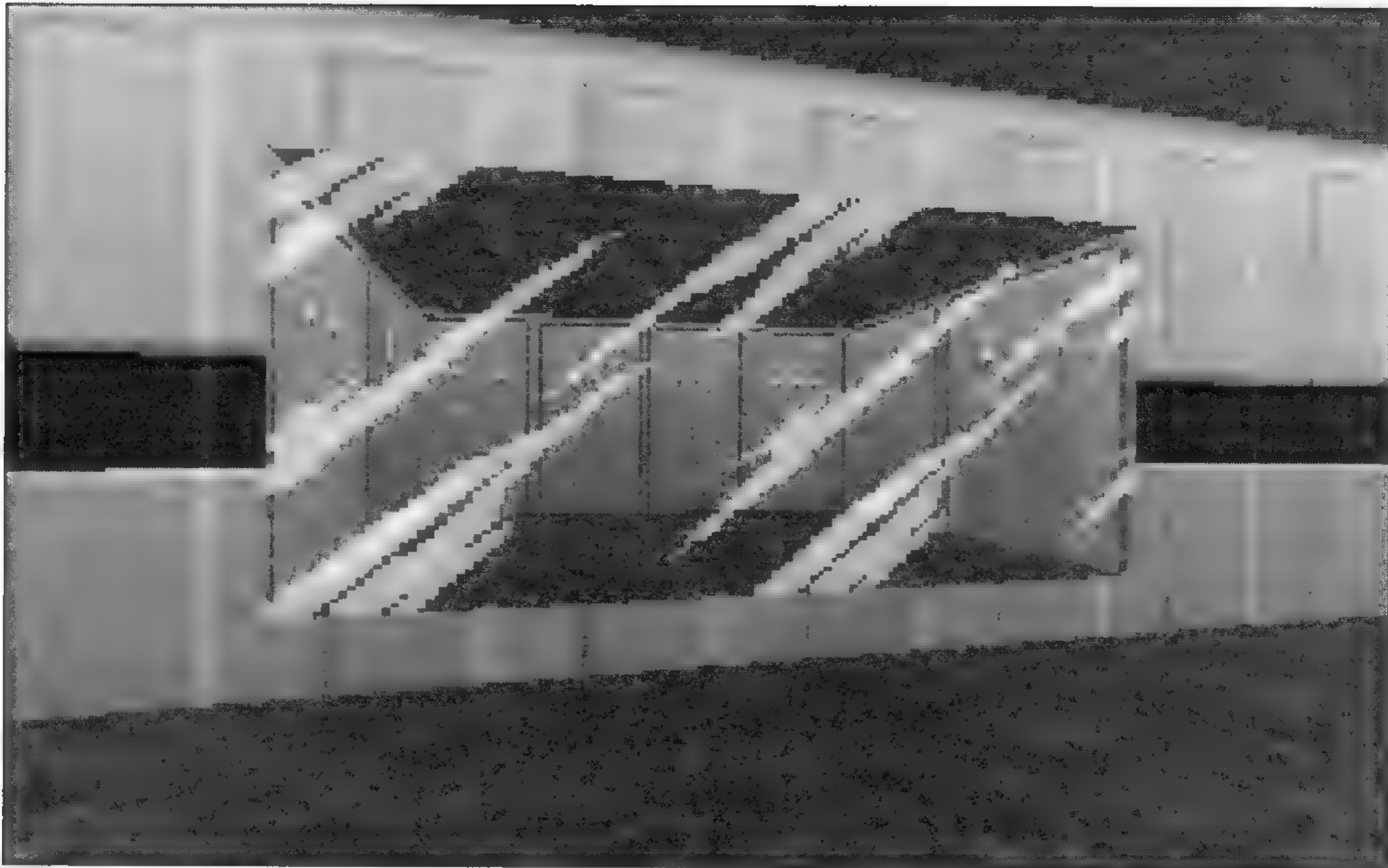


FIGURE 4.2: This masked two-sided wall has a glass texture applied to it.

place a texture on only *one* side of a two-sided wall, masking only one side of it. This is done exactly the same way, except you press Shift + M instead to mask the wall.

COPYING AND PASTING TEXTURES

Once you've found a good texture to apply on all the walls of a sector, you don't have to keep selecting that texture over and over using the texture selection screen. There is a nice texture copy-and-paste mechanism built into Build for copying textures from one wall and pasting them onto another. Follow these steps to copy and paste a texture:

1. Switch to 3D view mode.
2. Place the mouse cursor on a wall texture you want to copy.
3. Press the Tab key. This copies the texture into the Build *clipboard* (to borrow a Windows term).
4. Place the mouse cursor on the wall to which you want to paste the copied texture.

5. Press the Enter key (*not* the numeric keypad's Enter key, which will switch you to 2D mode). The texture in the clipboard will be pasted onto the selected wall.

Once a texture gets placed into the clipboard, you can paste it on as many walls as you want by moving the mouse cursor to each wall and pressing Enter.

You can also apply the texture in the clipboard to all the walls in a sector at one time by placing the mouse cursor on one of the walls and pressing Ctrl + Enter.

Replacing Textures in an Entire Level

You can paste the current texture in the clipboard on *every* wall in your level that has the same texture as the wall currently under the mouse cursor. For example, if several walls have the default brown brick texture and the mouse cursor is on one of these walls, you can replace the brown

brick texture on all the walls with the texture stored in the clipboard by using a single keystroke! To do this, place the mouse cursor on a wall and press Alt + C. Realize that if you press this keystroke accidentally, it could really mess up a level, so I *highly* recommend that you save the level first.

CONTROLLING THE PAINTING OF TEXTURES

When a texture is painted on a wall, and that wall is longer or higher than the actual length of the texture, the texture is repeated as needed. For example, if a texture is 128 pixels high, and a wall is 160 pixels high, then the texture will repeat 32 pixels in order to paint the wall completely.

By default, when you apply a texture to a wall, the top of the texture is placed at the ceiling of the sector, and the texture is applied or painted in a downward direction. If the wall is taller than the texture, the texture is repeated until the entire wall is painted.

In some cases, however, it is desirable for a texture to be painted beginning at the *bottom* of a wall, against the *floor*, and then upward toward the ceiling. For example, you will need to use this technique when you work with creating special door effects in chapter 7.



TIP

If you're applying the same texture to an entire sector, with a little practice you can get really good at spinning in a circle using the left or right arrow key and pressing Enter as each wall spins into view under the stationary mouse cursor.

You can change the drawing orientation so painting starts from the floor and works its way upward by putting the mouse cursor on the wall and pressing the O key. The O key acts as a toggle, so pressing it again with the mouse cursor on the same wall will revert the painting back to its original top-down orientation. Changing the painting orientation becomes important later, once you get your sectors moving, so it's a good idea to practice now.

To see this painting orientation in action, create a sector and put a texture on one of the walls. Choose a texture that has a discernible pattern, like texture #242, so you'll be able to see when it repeats. Make the ceiling of the sector taller until you definitely see the pattern of the texture repeat. You'll notice that as you raise the ceiling, the texture on the wall *sticks* to the ceiling, and more of the texture originates near the floor to paint onto the taller wall. Press the O key, and raise the ceiling a bit more. This time, you'll notice that the texture sticks at the floor, and the new texture is created near the ceiling. (See Figure 4.3.)

SCALING WALL TEXTURES

Sometimes, you will find the perfect texture for a wall that you're designing, but it's not quite the right size. An example of this might be that you design the perfect door,

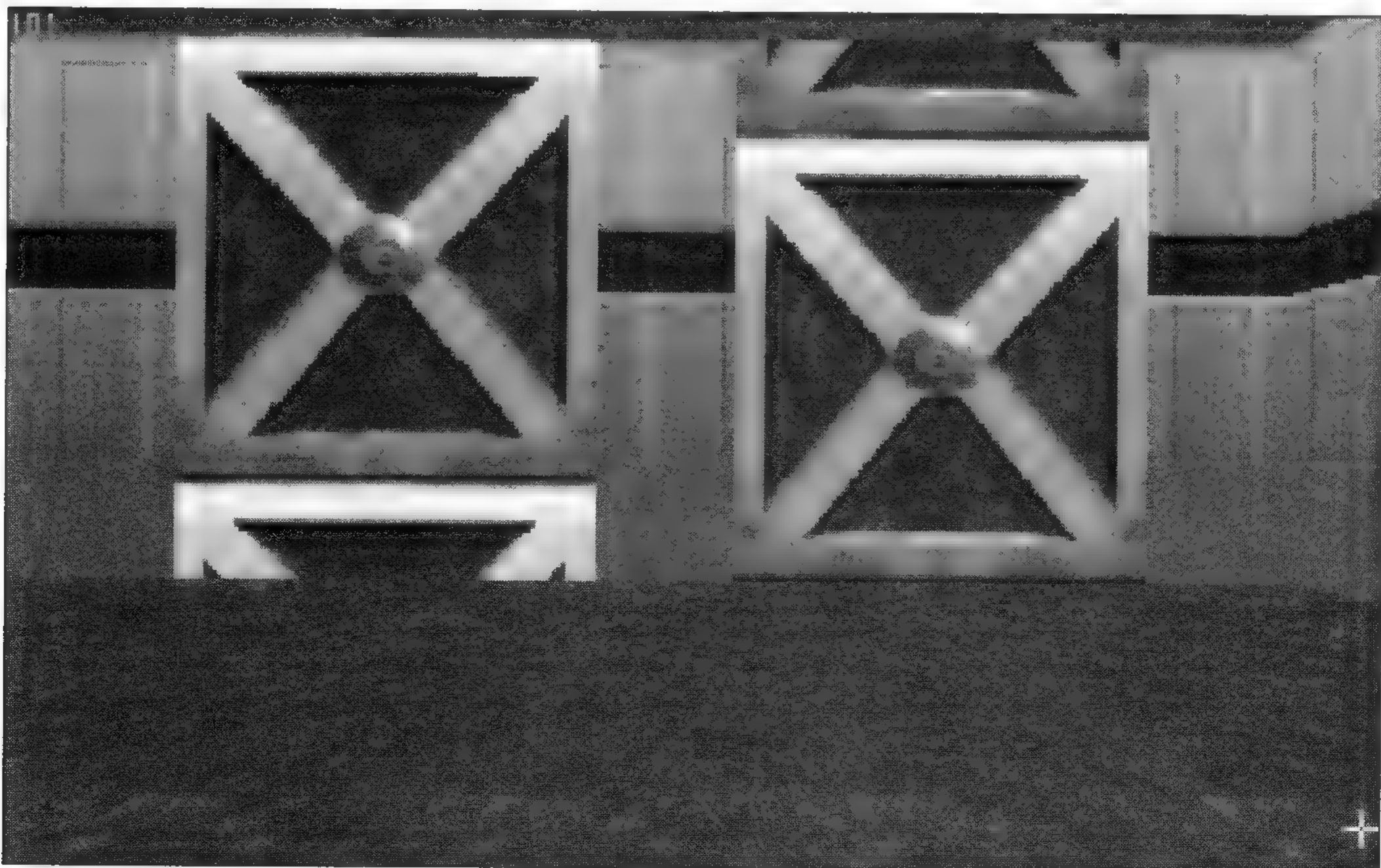


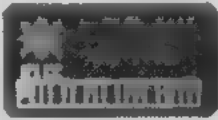
FIGURE 4.3: Top-down (left) vs. bottom-up (right) texture painting orientation is shown here.

perhaps a very small door that Duke will be able to enter only when he's shrunk by a nearby shrink ray. However, all of the door textures are too big for the door surface.

Textures can easily be *scaled* to any size. Scaling is done in 3D view mode. To scale a texture, place the mouse cursor on the wall with the texture you want to scale, and use the numeric keypad's 2, 4, 6, and 8 keys. Notice that these keys can double as a set of arrow keys: up, left, right, and down, respectively. Each time you press one of

these keys, the texture will shrink or stretch in the direction that the key's arrow points. You can also hold down the numeric keypad's 5 key while scaling, which will increase the amount of shrinking or stretching each time you press the numeric keypad's arrow keys (that is, the scaling will occur faster).

If you ever decide to return a texture to its default scaling properties, press the / (slash) key to reset the size back to the default setting. Figure 4.4 demonstrates the same texture on two walls with different scaling.

**NOTE**

If you scale a texture on a wall and then change to another texture, the new texture will take on the scaling attributes you applied to the first texture. Make sure to use the / (slash) key to reset the scale values for the new texture.

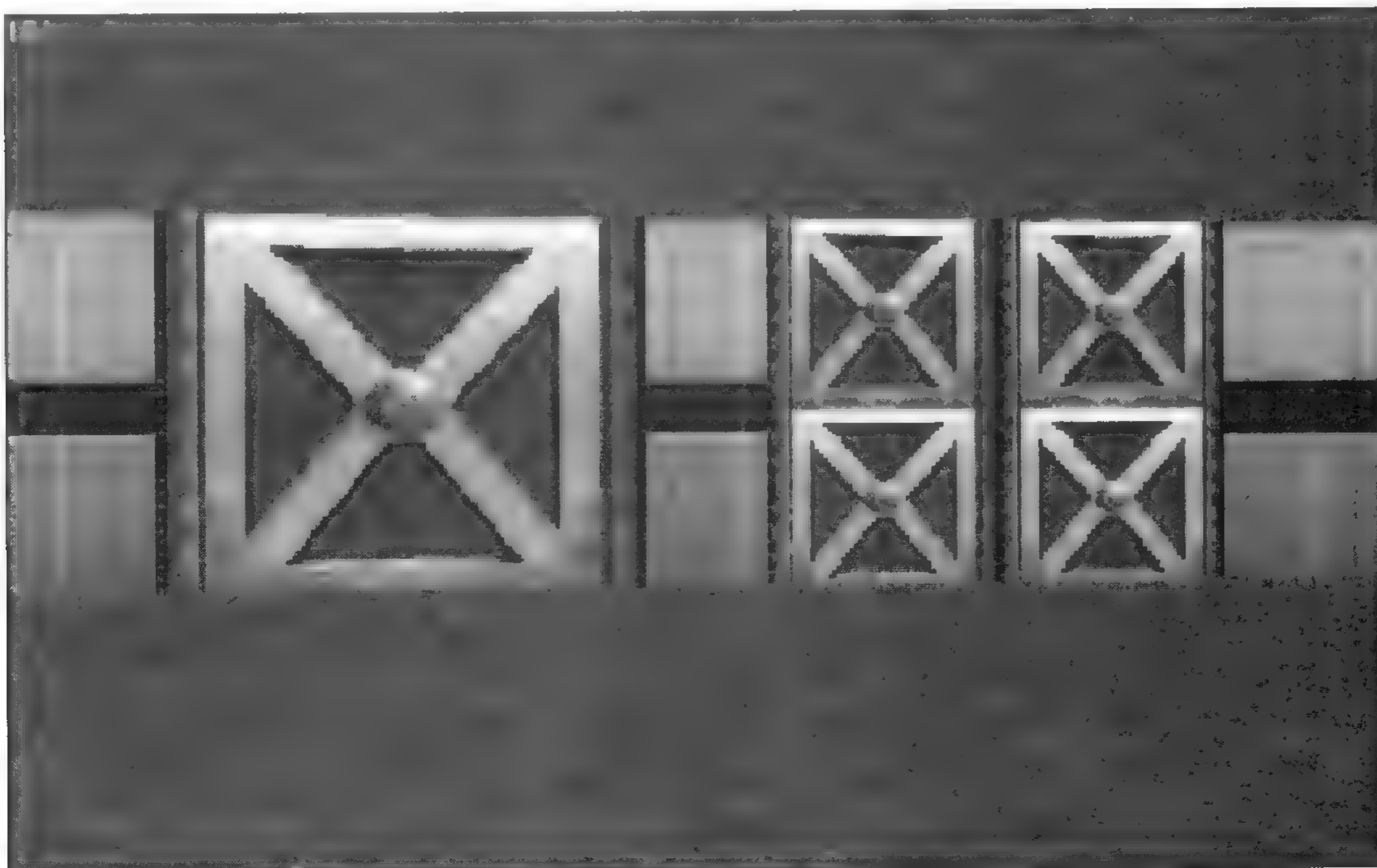


FIGURE 4.4: These two identically sized walls have the same texture. The left wall's texture is the default scale; the right wall's texture is scaled down to one-half the normal size.

PANNING WALL TEXTURES

You've already discovered that Build paints a texture (by default) from the top of the wall and paints downward. In addition, the left edge of the texture will also align with the leftmost side of the wall. This means that the upper left corner of every texture is mapped to the upper left corner of every wall. Depending on the length of your walls, this can cause misalignment of textures, which results in a *seam* where the two walls meet. You can eliminate this seam by *panning* the wall texture.

Panning is a technique for changing which point on the texture becomes the upper-left corner of each wall. If you think of the top-left corner of the texture as the origin on an imaginary coordinate axis, panning would be the act of sliding that origin to a new point on the wall.

Figure 4.5 shows two consecutive walls with the same texture. The leftmost wall isn't long enough to display the X-shaped beam a second time, so the texture is cut off at that point. When the second wall begins painting, the texture starts over again at the left edge, and the seam is visible.

To pan a texture, hold down Shift and use any combination of the numeric keypad's 2, 4, 6, and 8 (arrow) keys. The texture will pan in the direction of the key you

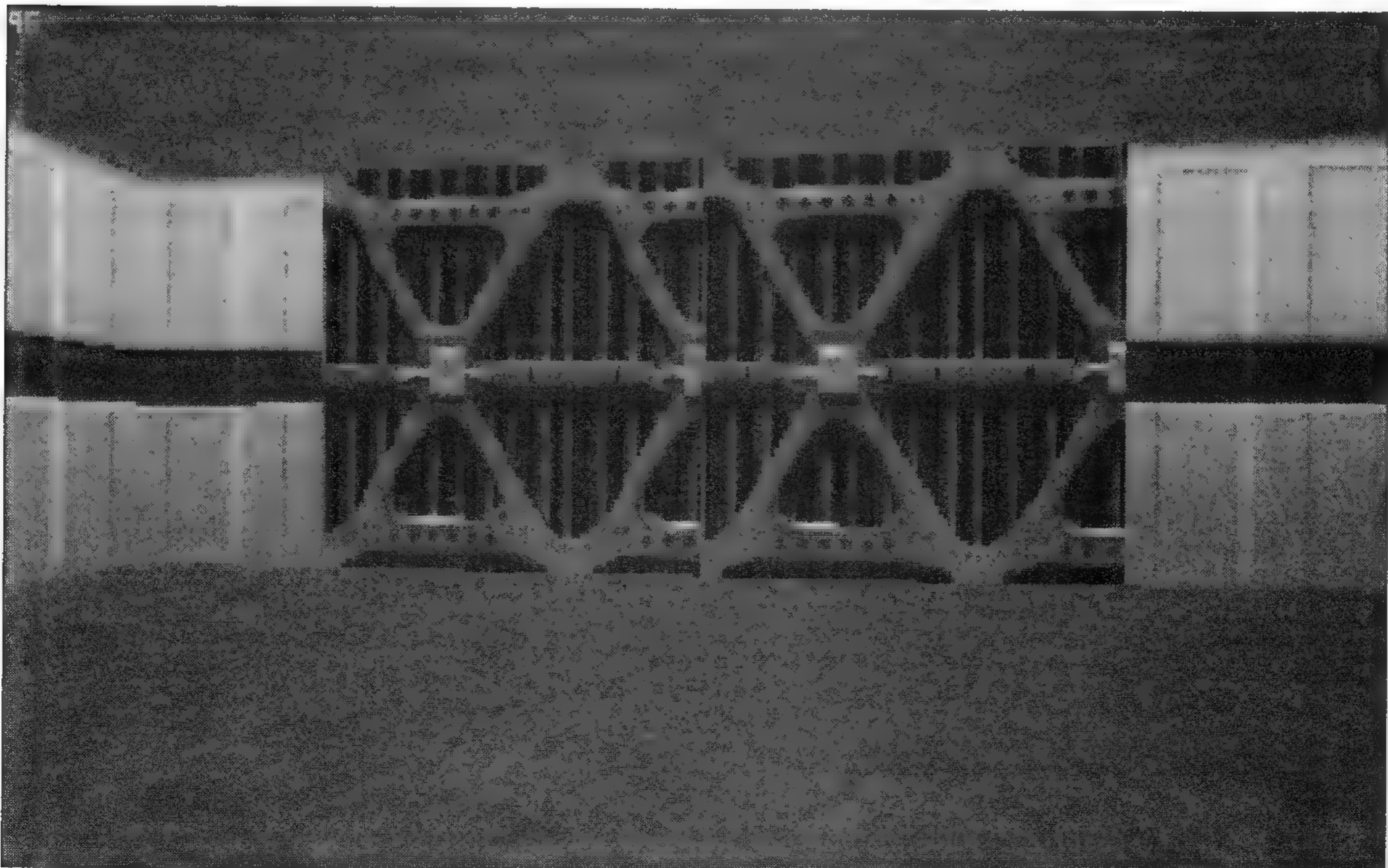


FIGURE 4.5: The result of not aligning your textures is a very unrealistic seam that becomes visible where the walls meet, making steel and rock textures look more like badly hung wallpaper.



press. Like texture scaling described previously, holding down the numeric keypad's 5 key while using the other keys will cause the panning amount to increase. Pressing the / (slash) key will reset the panning back to the default value.

ALIGNING WALL TEXTURES

Proper scaling and panning are crucial to the creation of a realistic-looking level. As mentioned above, the illusion of solid caves or large sheet metal walls is easily

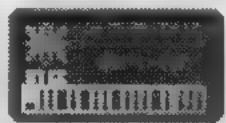
destroyed by a seam created by a badly aligned texture. Fortunately, there is an extremely useful and powerful function in Build that can eliminate these seams with a single stroke of the . (period) key.

To use this feature, choose a texture for the walls in your sector and place that texture on all the walls. (Don't forget the texture copy and paste function for quickly accomplishing this.) Apply any scaling properties you want to *one* of sector's walls. Once the texture is exactly how you want it, press the . (period) key, and

all the walls of the sector will be automatically scaled and panned to the first wall.

Build's algorithm for automatically aligning the textures starts from the first wall and moves in a straight line to the right. If a wall is encountered that doesn't have the same texture number, the process stops. So, if you want to align an entire sector's wall textures, make sure each wall has the same base texture number.

Texture alignment is generally one of the last steps when creating sectors because moving a single vertex or adding a new wall will usually destroy the alignment.



NOTE

Just like scaling textures, panning values do not reset on a wall when you change to another texture, so you may have to press the / (slash) key after adding a new texture to a wall to reset the panning values.



WARNING

I have found that the auto texture alignment function occasionally hangs a system. Thus, it is a good idea to save the level just prior to using this function.

EMBELLISHING CEILINGS AND FLOORS

You have learned how to apply textures to walls and then modify them; now you will learn how to apply textures to ceilings and floors. Most of the techniques for working with textures on ceilings and floors are exactly the same as those for walls. In fact, most wall textures can be used on ceilings and floors. However, there are enough differences between working with these types of surfaces that separate discussion is necessary.

CHANGING CEILING AND FLOOR TEXTURES

Changing a texture for a ceiling or floor is done the same way as it is for wall textures, that is, by placing the mouse cursor on the surface you want to paint while in 3D mode, and using the V key to bring up the texture selection screen. After pressing the V key, you will be presented with a screen showing all textures used on ceilings and floors in your level thus far, sorted in the reverse order of the frequency of their use in the level. Pressing the V key again brings up the display of all available textures.



NOTE

A texture must have a height and width that's an exact multiple of 8 pixels to be used on a floor or ceiling. If you choose a texture that's not a multiple of 8, you'll know it immediately because the texture will look very strange.

Copying Textures for Ceilings and Floors

Copying and pasting textures for ceilings and floors works the same way that it does for walls. In fact, you can copy a texture from a wall and paste it on a ceiling or floor. In fact, you can copy a texture from any surface—wall, ceiling, or floor—to any surface.

SCALING CEILING AND FLOOR TEXTURES

Scaling ceiling and floor textures is actually easier than it is for wall textures because there are only two possible sizes available, the normal size and a compressed size.

The key for toggling between these states is the E key (think of E for Expand). Press this key with the mouse cursor on a ceiling or floor, and you'll see the texture toggle between normal size and half-normal size. Figures 4.6 and 4.7 show the same texture on a floor in each of its available sizes.

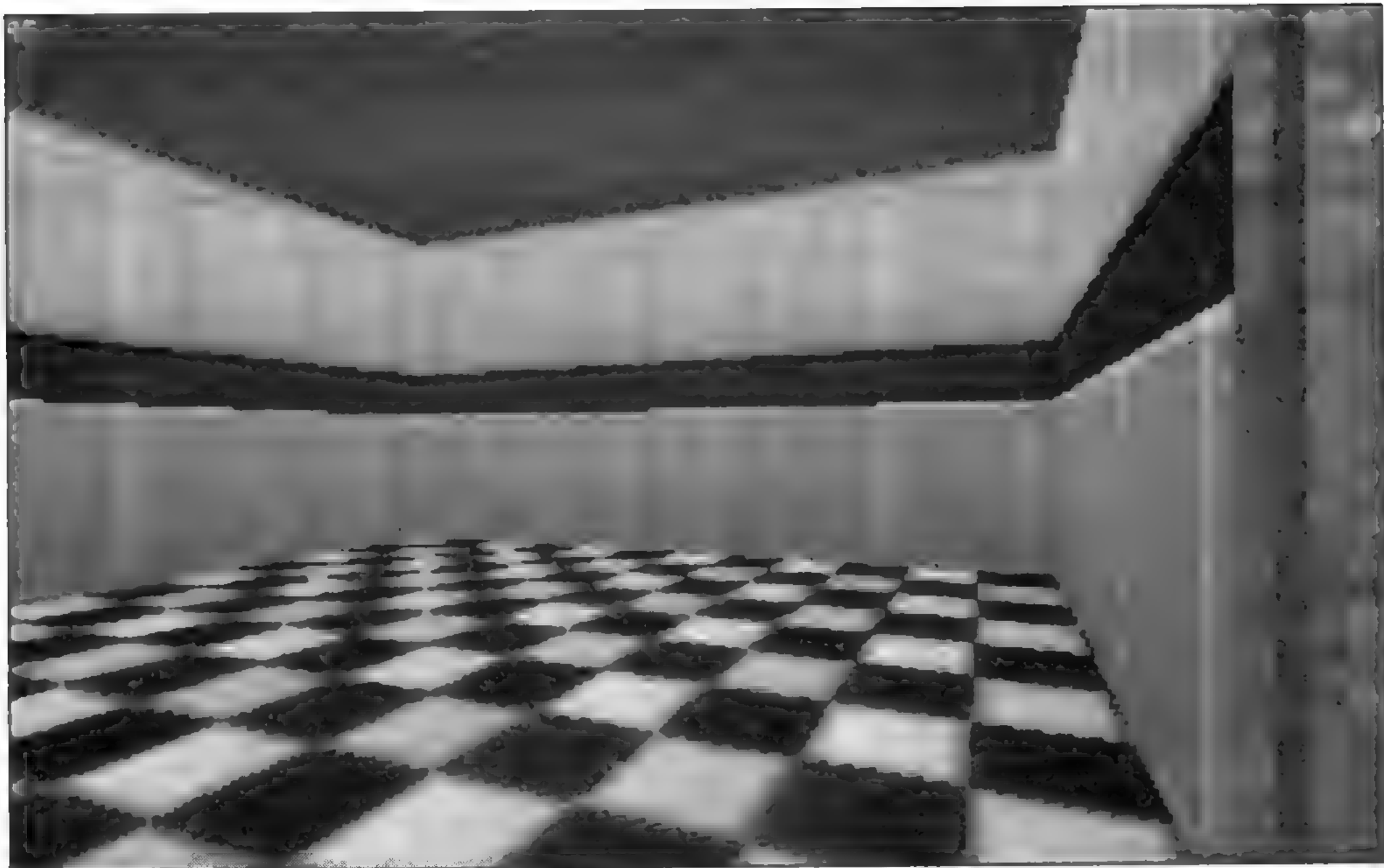


FIGURE 4.6: This is a floor texture at normal size.

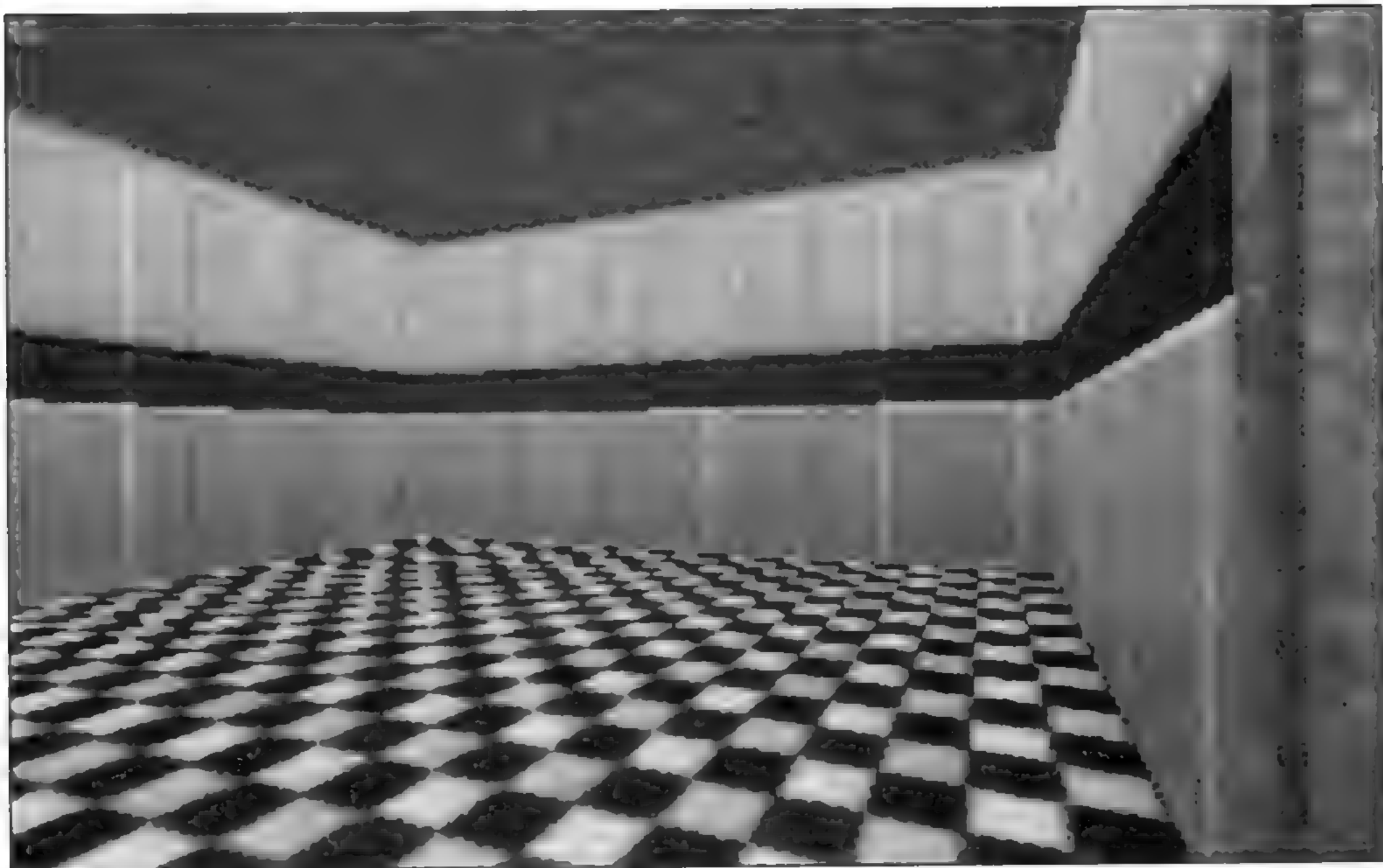


FIGURE 4.7: The is the same floor texture you see in Figure 4.6, but it is a compressed size.

The ability to compress ceiling and floor textures effectively doubles the amount of available textures for you to choose from. Make sure you keep this in mind as you're searching for new and unique ways to paint your level.

PANNING CEILING AND FLOOR TEXTURES

Ceiling and floor textures are painted onto their surfaces a bit differently than wall textures. The origin for a ceiling or floor texture is an absolute coordinate value somewhere on the map. This usually means that consecutive sectors with the same floor or ceiling textures will align themselves automatically. However, when placing a texture such as a panel of lights onto a ceiling, you may find that you will have to pan the texture to get the light panel texture to align properly to the sector. Figure 4.8 shows a badly aligned light panel texture, corrected in Figure 4.9.

Panning ceiling and floor textures is done in a similar way that it's done for walls, that is, by using the numeric keypad's 2, 4, 6, and 8 (arrow) keys. (The Shift key is not required as it is for wall texture panning.) You can hold down the numeric keypad's 5 key while using its arrow keys to increase the panning increment.



FIGURE 4.8: This is a badly aligned light panel texture.



FIGURE 4.9: The same light panel texture you see in Figure 4.8 was panned here so it fits correctly.

RELATIVE MAPPING

By default, all ceiling and floor textures are oriented with their upper-left corners in the northwest end of the map, and they are repeated as needed. Many times it is desirable to rotate a ceiling or floor texture so its pattern aligns with the edges of a sector. This becomes especially true with sectors drawn with diagonal lines. Any pattern on a texture placed on a ceiling or floor will be at a strange angle to the diagonal lines that make up the sector.

You can change the orientation of the ceiling and floor textures by pressing the R key. The new orientation will be along the *first wall* of the sector. Recall that the first wall of a sector, which was discussed in chapter 3, is the first wall that you draw when you create the sector, but you can change which wall Build considers the first wall by using the Alt + F key in 2D or 3D view mode. Orienting a texture to the sector's first wall allows the pattern on the texture to “line up” with the lines that define that sector. Making a ceiling or floor relative will also *lock* the texture onto the surface, so if the sector moves, the texture will move with it. You will notice this when you start creating moving sectors later. Figures 4.10 and 4.11 show the difference between nonrelative and relative floor textures.

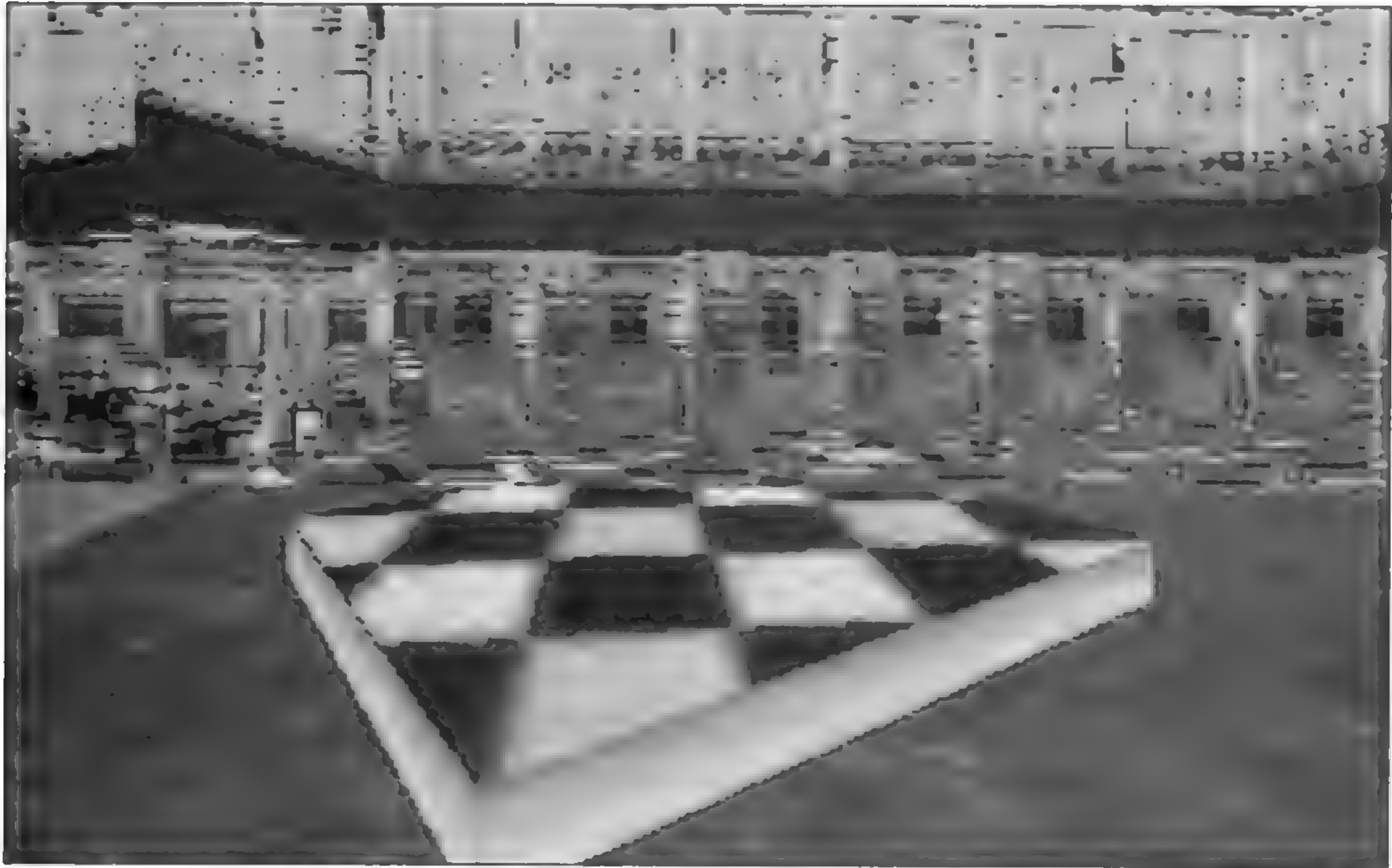


FIGURE 4.10: This is a sector with a floor texture that is not relative.

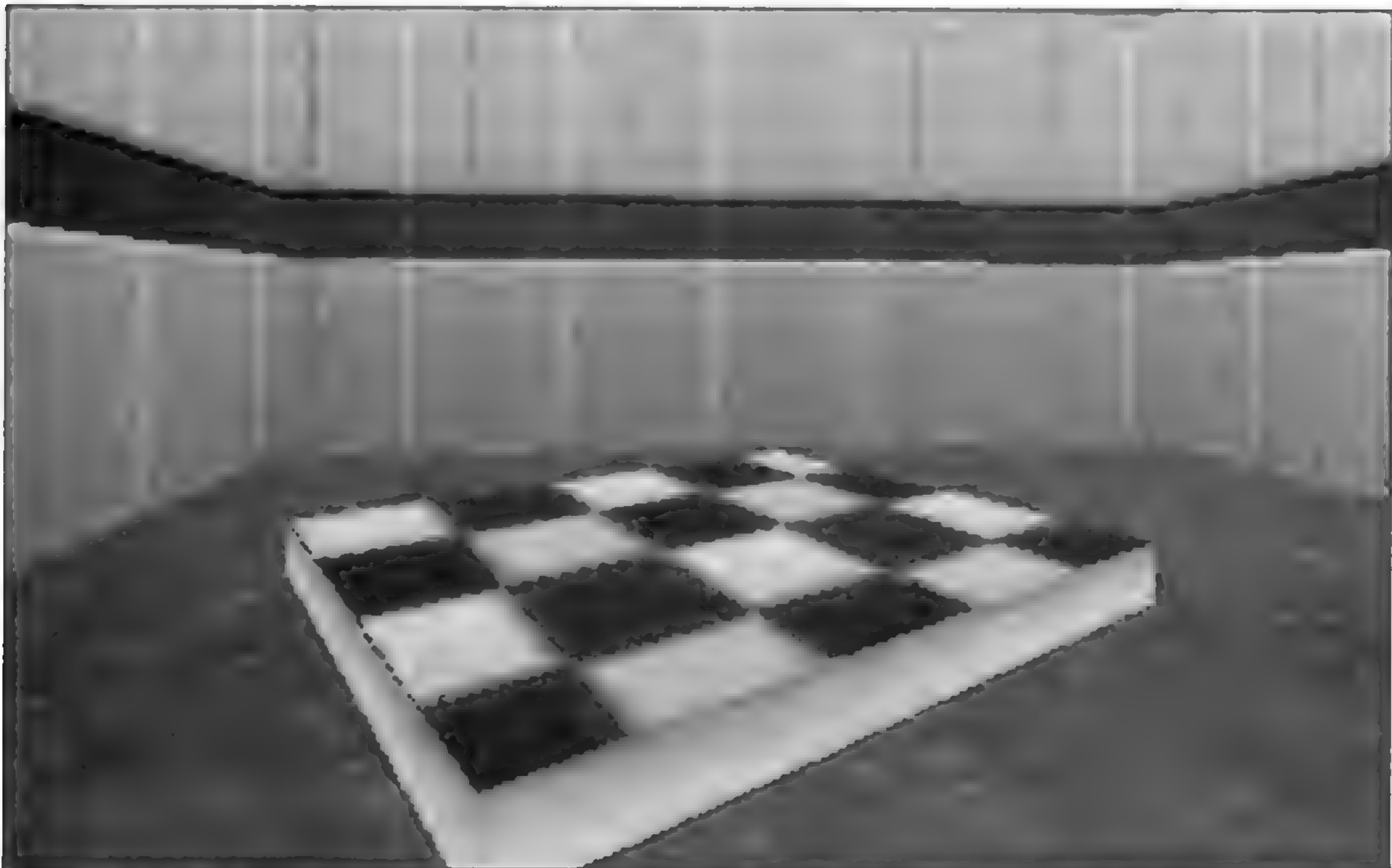


FIGURE 4.11: This is the same sector you see in Figure 4.10 with the floor texture aligned relative to the sector's first wall.

PARALLAXING TEXTURES

The Build engine creates the illusion of a large expanse of sky on the ceiling of a sector by using a different painting method. This alternate method of texture mapping is known as *parallaxing*. To see the difference, make a sector of any size, and put one of the sky textures on it, like texture #89 (called LA), as in Figure 4.12. The sky won't look correct at all until you parallax it, which you do by pressing the P key with the mouse cursor on the ceiling. Figure 4.13 shows the parallaxed version. For a strange effect, you can also parallax the floor, which is done in the sectors that look like outer space in Episode 2.

There are three different types of parallaxing, which you can toggle by pressing Ctrl + P with the mouse cursor on the sector in 3D mode. The three types are normal, compressed, and curved. Experiment with the three types to see which looks best in your particular environment.

This brings up an interesting consequence of a parallaxed sector. If you create any parallaxed ceiling with any of the BIGORBIT textures (#84–88), Duke will die *instantly* when he enters that sector. This will happen *even* if the God mode cheat is enabled. You can disable this *automatic-death* feature by placing a palette on the ceiling, which will be discussed shortly in this chapter.

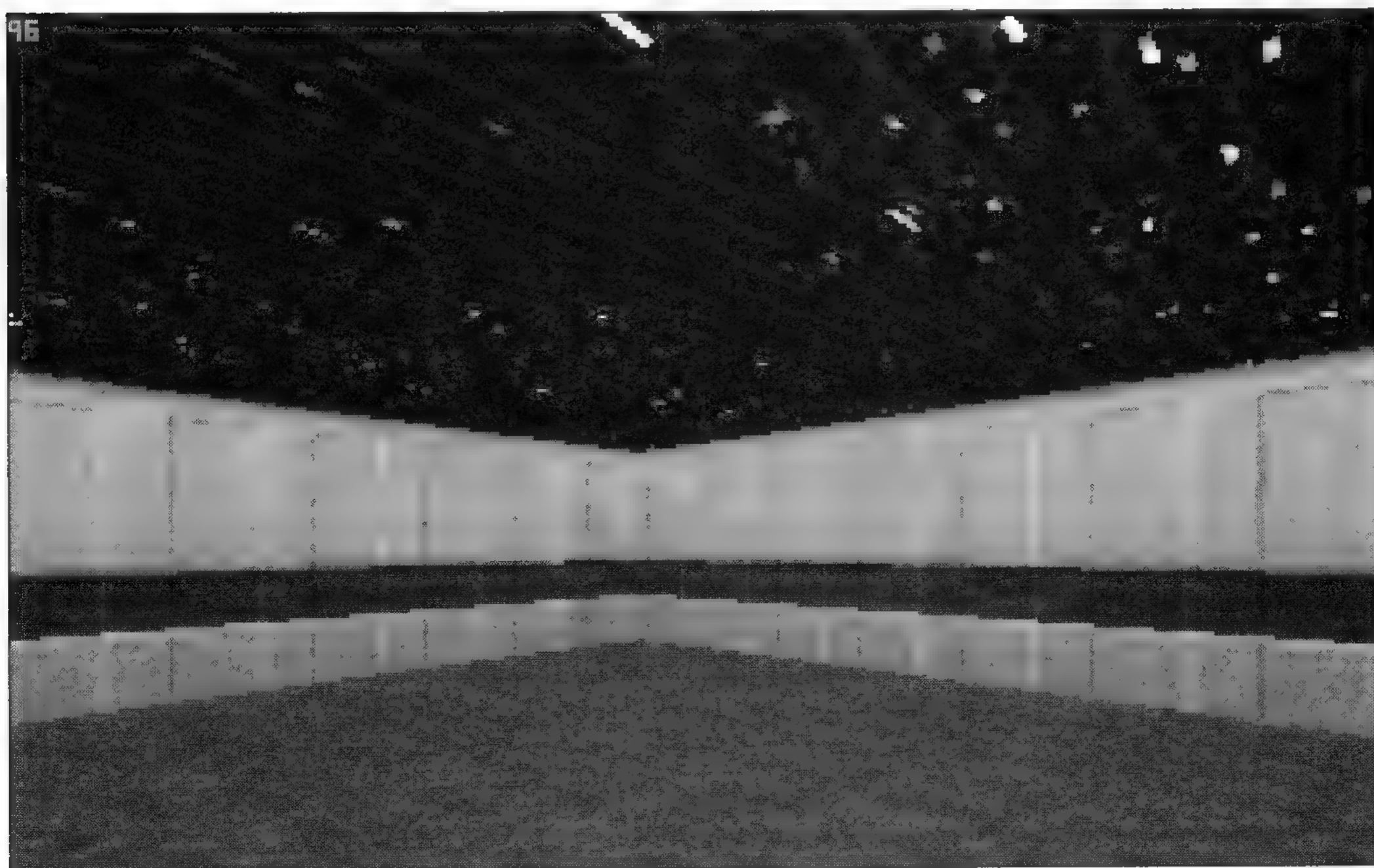


FIGURE 4.12: This is a ceiling with texture #89 that is not parallaxed.

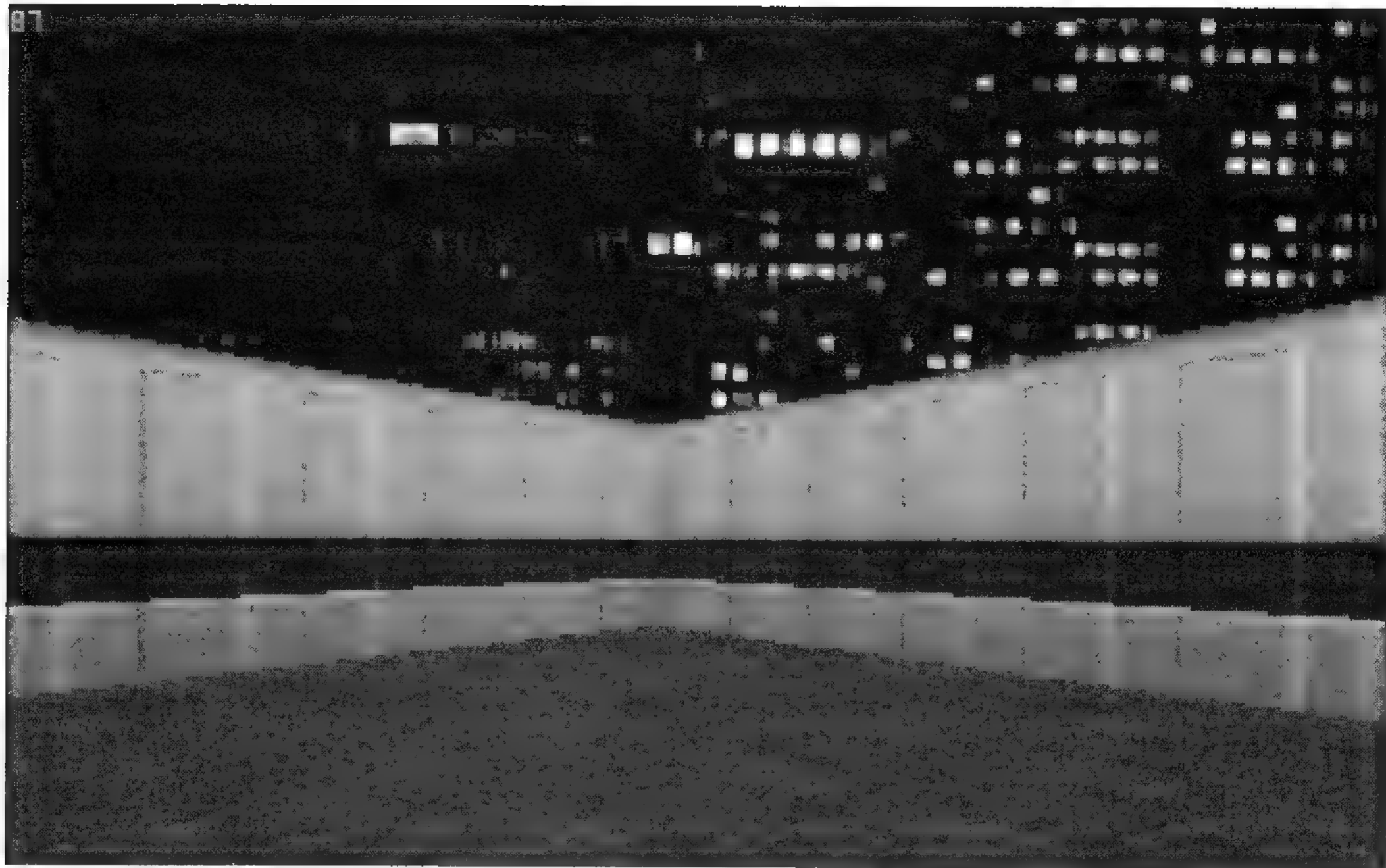


FIGURE 4.13: The same ceiling texture you see in Figure 4.12 has parallaxing enabled here. Now it looks like the city skyline the designer intended.

APPLYING SHADE

The proper application of textures is critical to creating a believable level. However, even after you apply the perfect textures to each and every sector of your level, it may still look a bit flat. The next consideration is setting the proper lighting of the level. Proper lighting further enhances the level by making it appear like a more realistic place.

Like applying texture, the mechanics of applying shade are very easy, but the realistic application of shade is what will separate the good level from the bad level. So, let's get those mechanics out of the way, and plunge right into a few examples. Shading a wall, floor, or ceiling is done in 3D view mode by placing the mouse cursor on the surface and pressing the numeric keypad's + (plus) and - (minus) keys. The + (plus) key lightens the selected surface; the - (minus) key darkens it. To darken or lighten an entire sector, simply apply the shade value to each surface. You can also type a shade value by pressing the ' (apostrophe) key along with the S key (i.e., ' + S).

When you use the Tab and Enter keys for texture copying and pasting, the shade value of the source surface will copy along with the texture into the texture clipboard, so it's a great shortcut to get one wall of a sector textured and shaded exactly the way you want, and then paste those attributes onto every other wall of the sector. In addition, by pressing Shift + Enter when pasting a texture onto a new surface, you will paste *only* the shade value, leaving the existing texture intact. This feature is great for shading the ceiling and floor without affecting the textures you've already applied there.

Complex lighting effects can be created by the clever application of shade. Consider Figures 4.14 and 4.15, which show the 3D and 2D views of the same room. Look first at the 3D view. Five total sectors make up the room, but from 3D mode, only two sectors are discernible by looking just at ceiling and floor elevations. However, look more closely at the shade of the ceiling and floors. Sectors have been carefully arranged near the raised platform so shading could be used around the platform. When used together with the texture of the light you see on the far wall, the effect is a believable illumination onto the platform, causing the platform to cast a shadow. Creating these extra sectors, simply for the purpose of lighting, is the difference between a cool level and just an OK one.



FIGURE 4.14: This is a 3D view of a room using shade and extra sectors to create shadows.

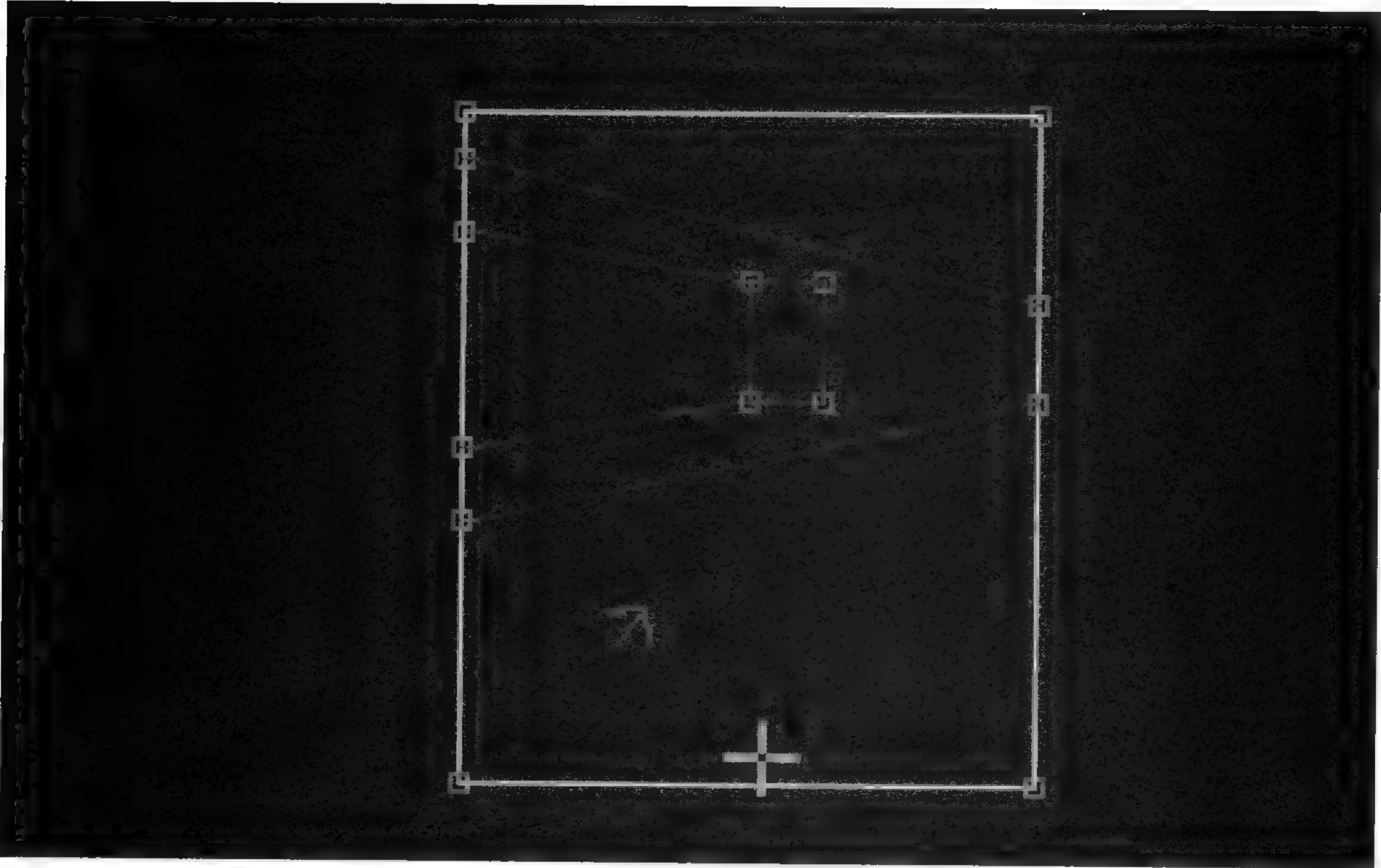


FIGURE 4.15: This is a 2D view of the room shown in Figure 4.14.

APPLYING PALETTE

Like shade, the *palette* of a wall, floor, or ceiling can be used to significantly alter the appearance of a room. The palette of a surface refers to the set of colors that are used when drawing the surface. The default palette is 0, which means to use the normal colors that appear in the texture itself. However, there are alternate palettes you can use to give a sector the appearance that colored lights are illuminating it. The palettes available for walls, floors, and ceilings are shown in Table 4.1.

| TABLE 4.1: PALETTE TABLE | | |
|--------------------------|----------|-------|
| NUMBER | PALETTE | NOTES |
| 0 | standard | |
| 1 | blue | |
| 2 | red | |

(Continued on next page)



(Continued from previous page)

| NUMBER | PALETTE | NOTES |
|--------|---------------------|---|
| 3 | standard | The difference between this palette and palette 0 is that using this one on a parallaxed BIGORBIT ceiling will prevent the sector from automatically killing the player upon entry. This palette also prevents a subway sector with a parallaxed ceiling from firing rockets at a player. |
| 4 | black | |
| 6 | night-vision colors | |
| 7 | brown | |
| 8 | green | Changes water to toxic sludge or slime. |

Applying a palette to the surfaces of a sector can make the entire sector appear as if it's bathed in a colored light. For example, to give an entire room a blue hue, simply give all the walls, floors, and ceilings a palette of 1.

To apply a palette, switch to 3D view mode, place the mouse cursor on the surface to color, and press Alt + P. Then, type the palette number shown in Table 4.1.

A good example of color palettes in action is the dance hall on The Red Light District level in L.A. Meltdown (E1L2). Take a visit there and note the great use of both the red and blue palettes. Note that these sectors are also a dark shade, as you would expect a bar to be. This area is a great example of texture, shade, and palette all coming together to create a realistic-looking place.

EXAMPLE APPLICATIONS OF TEXTURE, LIGHT, AND PALETTE

Let's look at several examples of the effective application of texture, shade, and palette that can be found in Episode 2 of *Duke Nukem 3D*, Lunar Apocalypse.

CASE STUDY #1: THE DARK SIDE (E2L8)–COURTYARD AREA

I really enjoy one of the first views you see in this level, shown in Figure 4.16. The open courtyard area holds several of the eggs that spawn the Protozoid Slimer creatures. This level features twelve sectors arranged in a radial pattern to form a twelve-sided polygon, as shown in Figure 4.17. The ceilings of each sector alternate in texture

from a normal blue and beige striped pattern (#217) to the parallaxed MOONSKY1 texture (#80). Also, each of the twelve sectors' ceilings are sloped toward the center, giving a domed roof effect. The walls of this sector are very low in height, but the high sloped ceiling, together with the parallaxed MOONSKY1 texture beyond, really gives the area the illusion of space. Every other outer wall of the basic room is masked and transparent to allow a clear view to a jagged-shaped outer sector beyond.

This jagged outer sector uses textures to make it look like the lunar surface. The ceilings are again textured with MOONSKY1 and parallaxed. The walls of this sector display texture #238, a gray rough-looking rock texture. Finally, the floor of the sector is another gray texture, #478. This texture looks like small gray rocks or pebbles.

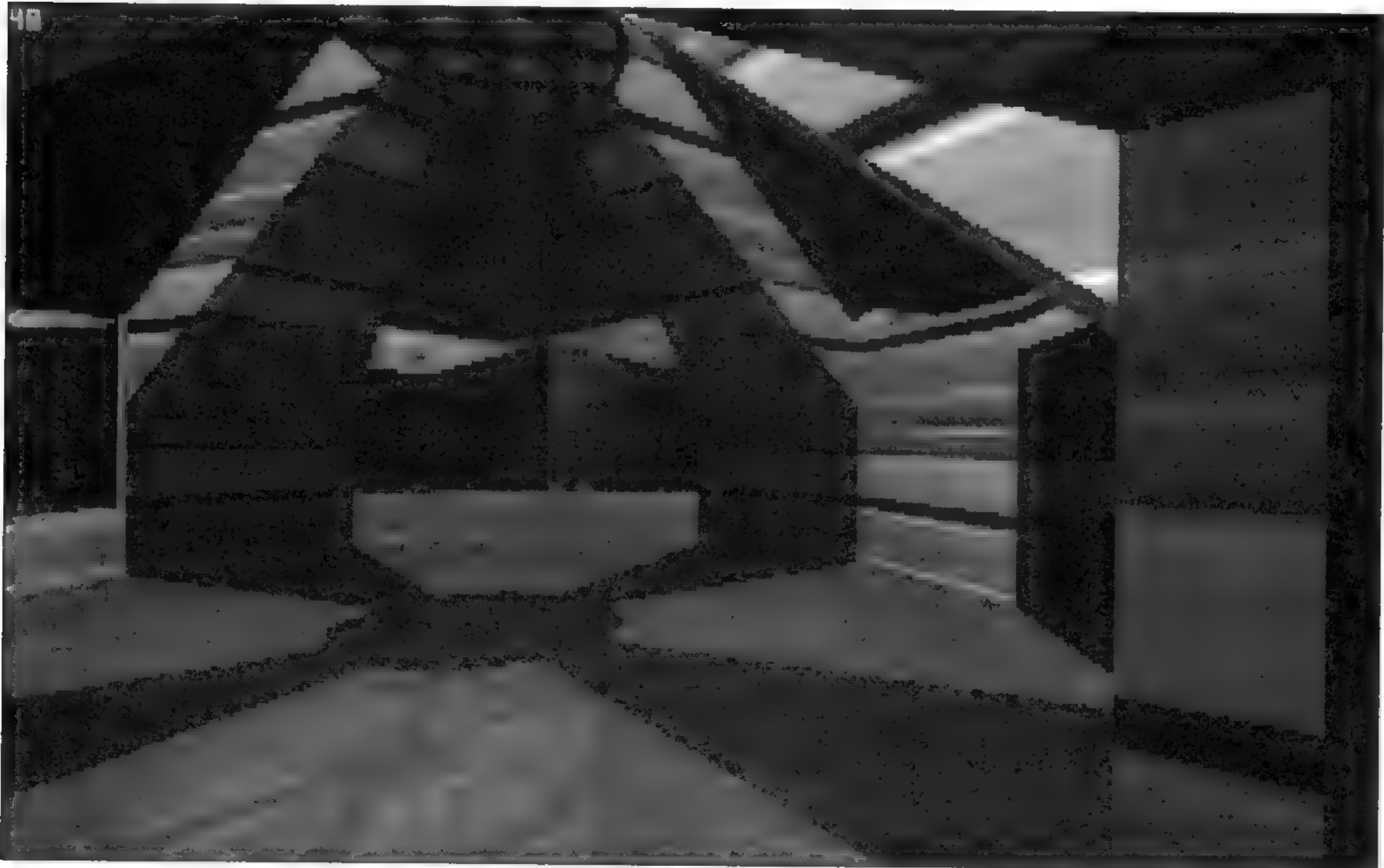


FIGURE 4.16: This is the open courtyard hatchery in 3D view mode.

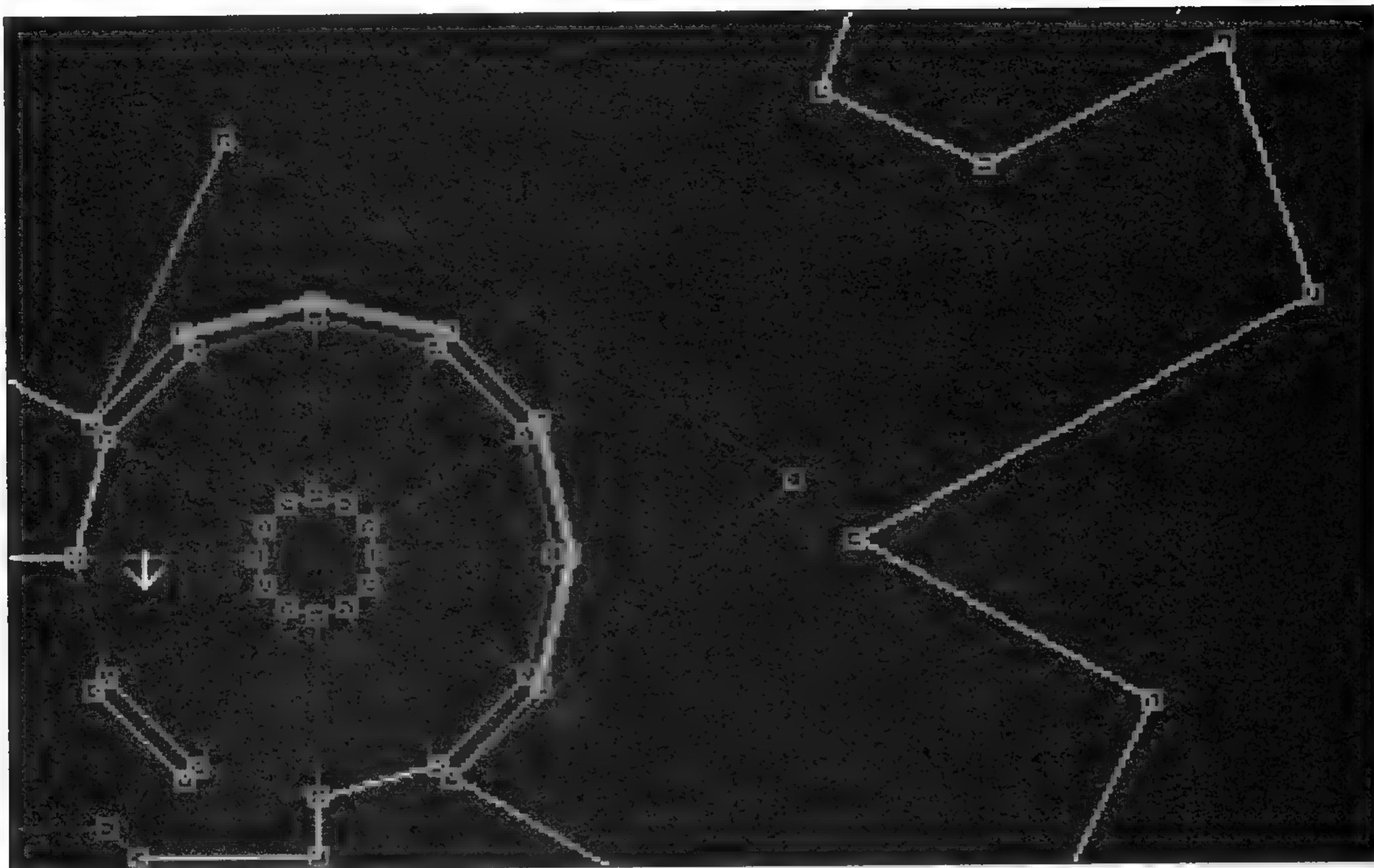


FIGURE 4.17: The open courtyard is shown here in 2D view mode.



FIGURE 4.18: This is a 3D view of an area full of strange machinery on The Incubator (E2L2) level.

CASE STUDY #2: THE INCUBATOR (E2L2)–BLUE AREA

This example shows the effective use of palette and textures to create a mood. Figure 4.18 shows the northwest corner of the map, if you want to look at it in Build. The overall area has a blue color that appears to emanate from the rotating devices on the west side of the room.

Texture #242 is a pair of small blue rectangular lights that makes a larger wall look like it's completely covered with these small lights. This texture is used around the rotating devices. The walls near this are a metallic texture (#389) that already has a blue tint to it. The floor of the western sector that houses the rotating objects is a standard gray color, but it is given a blue palette to enhance the illusion of the area being bathed in blue light. The ceiling of the same sector uses texture #707, which looks like an overhead grid of fluorescent lights, but it too is given a blue palette to suggest that more blue light is coming from above. Finally, near the edge of this sector, thin walls are given the animated texture SLIMEPIPE (#538), which looks like vertical pipes carrying some type of liquid. The overall color of this texture is also blue.

I like this area as a demonstration of palette use because certain textures were chosen that already contained colors that contributed to the overall color scheme of the area, and color palettes were applied to other textures to make them the right color. Figure 4.19 shows a 2D view of this area.

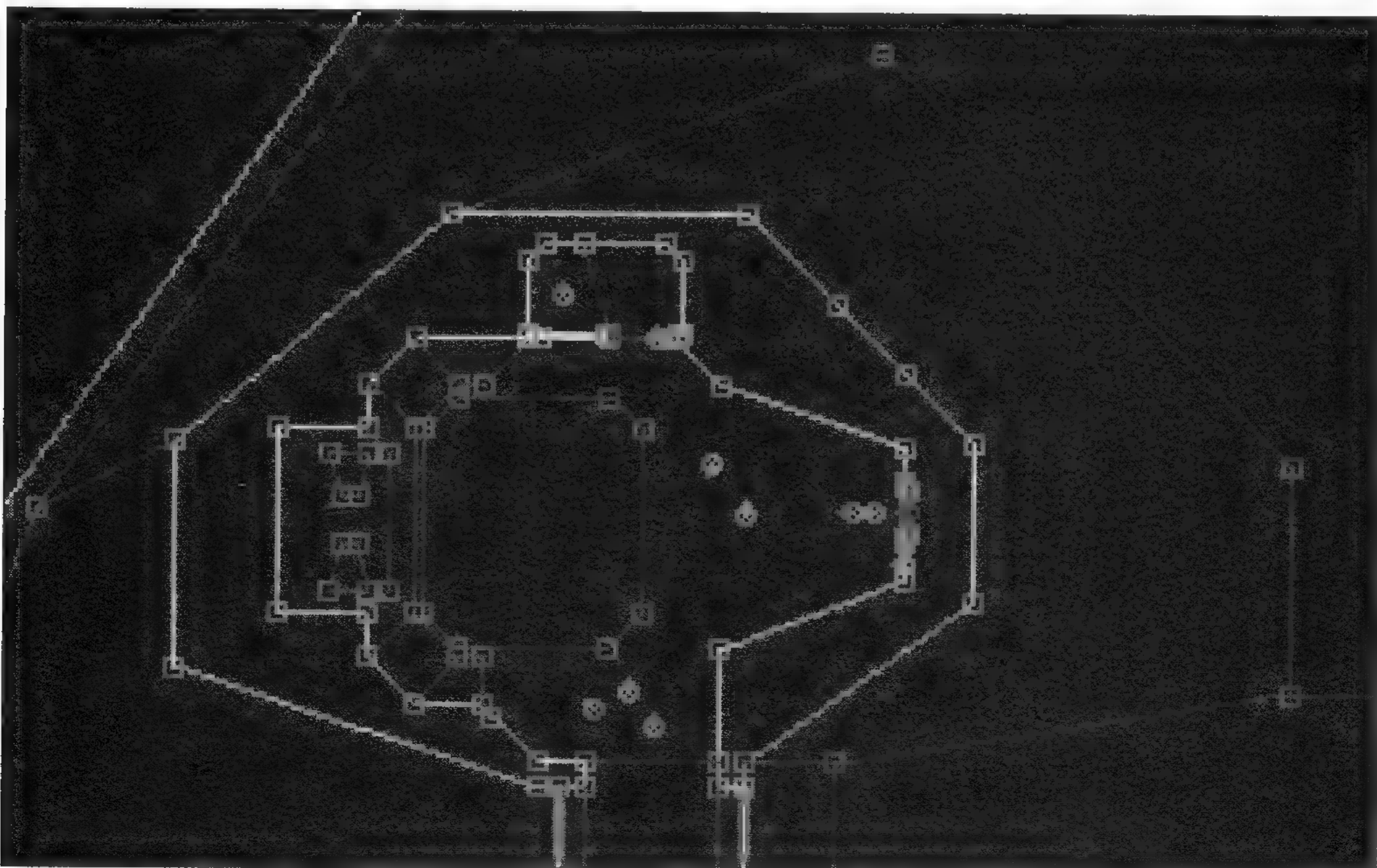


FIGURE 4.19: The strange machinery area is shown here in 2D view.

CASE STUDY #3: THE DARK SIDE (E2L8)–MONOLITH AREA

When I first encountered this area in the game, I was literally spooked. The combination of light and darkness, along with the sound effects, makes this area nothing short of eerie, as you can see in Figure 4.20. This is possibly one of the most memorable areas in terms of sector design that I can recall. The area consists of a hollowed-out cave. A bright light shines toward a strange monolith. The monolith acts as a teleporter to another area of the map.

The cave itself is created using texture #240, a rough rock that is usually used for blown-out holes in walls. The monolith is a specialized texture created just for this room. It is a smooth black texture with a bright corner to simulate a reflection.

The shading of this area is one of the things that makes it memorable. The contrast between light and dark is very high. That is, the dark sectors are *very* dark, and the bright areas are *very* bright. The sectors are constructed so that the wall with the light panel appears to be the only light source in the sector. Extra sectors are created behind the monolith to look like sharp shadows being cast by the light panel. In addition, the designer took extra time to create bright and dark sectors in the west hallway



FIGURE 4.20: The strange and eerie monolith is shown in 3D view.

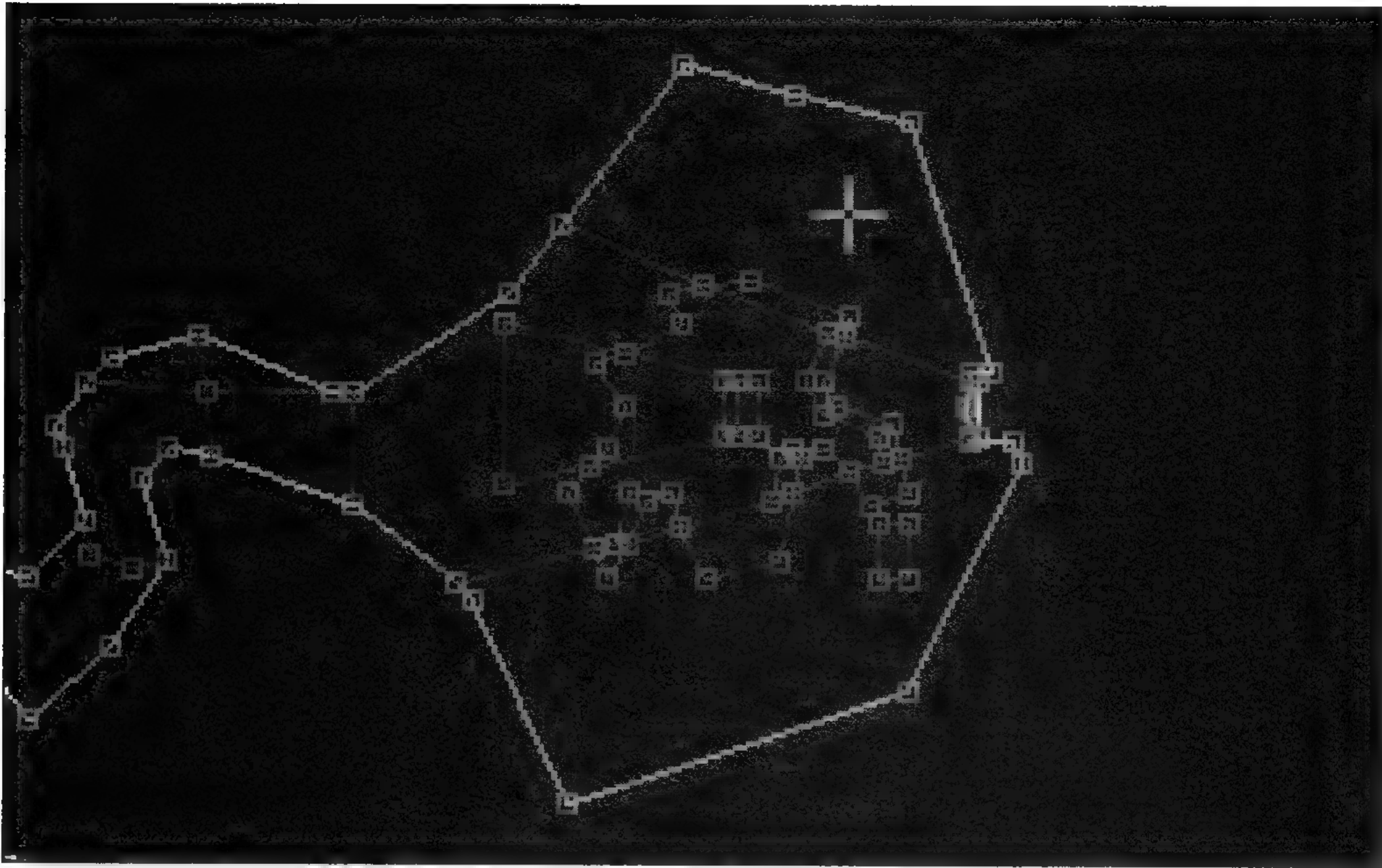


FIGURE 4.21: Here's the monolith room shown in 2D view.

leading out of the room, as you can see in Figure 4.21. This shows how even a simple hallway can be made more exciting with realistic lighting.

ONE FINAL WORD ON SECTOR DESIGN

If your textures are not properly aligned or are badly chosen, and every sector has the same brightness and color, then your level will be boring—period. Players may not know exactly *why* they think it is so dull, but they will know it does. One of the reasons the original *Duke Nukem 3D* levels look so awesome is because of the excellent use of texture, light, and shade. As you download and play the hundreds of home-made levels that become available, pay attention to these details. Chances are, if you finish playing the level and think to yourself, “Wow, well-done level!” then that level designer made good use of these features.



KEYSTROKE AND OPERATION SUMMARY

A summary of all of the keystroke sequences and operations covered in this chapter appears in Table 4.2.

TABLE 4.2: CHAPTER 4'S KEYSTROKES AND OPERATIONS

| KEY SEQUENCE/ OPERATION | FUNCTION | MODE PERFORMED |
|------------------------------------|---|-------------------|
| 2 | Allow top and bottom of wall to have different textures | 3D |
| M | Create a masked wall (both sides) | 3D |
| Shift+M | Create a masked wall (one side) | 3D |
| Alt+C | Paste wall texture in clipboard onto <i>all</i> walls on level with current texture | 3D |
| O | Change painting orientation of wall texture (top-down vs. bottom-up) | 3D |
| 2, 4, 6, and 8 (numeric keypad) | Scale current wall texture or pan current wall, ceiling, or floor texture | 3D |
| / | Reset wall texture scale or panning to default | 3D |
| . (period) | Align all like wall textures in a loop automatically | 3D |
| E | Scale texture (two possible sizes) | 3D |
| R | Relative-map ceiling or floor texture | 3D |
| P | Parallax ceiling or floor texture | 3D |
| Ctrl+P | Change parallax mode for ceiling or floor | 3D |
| +/- or '+S | Shade wall, ceiling, or floor | 3D |
| Shift+Enter | Paste shade value of texture in clipboard onto current wall, ceiling, or floor | 3D |
| Alt+P | Change wall, ceiling, or floor palette | 3D |





Placing Objects
with Sprites

Duke Nukem 3D is dubbed as the latest in 3D game technology, and for the most part, this is absolutely true. The structures you create as the level designer appear to the game player as a true three-dimensional environment (with a few exceptions). However, the *objects* that exist in *Duke Nukem 3D* levels are not 3D objects at all. In fact, each one of the objects you see in the game is strictly a *two-dimensional* bitmap. These 2D objects are placed into the 3D world, and because the bitmaps are cleverly drawn to appear 3D, you experience the game as if a 3D creature is following you around or bearing down on you menacingly.

WORKING WITH SPRITES

All of the objects you see during play in *Duke Nukem 3D* are present because of the placement of *sprites* by the designers. The word *sprite* is a computer term that refers to a picture that represents a solid object placed into a game. This definition certainly holds true in this case: All of the objects in the *Duke Nukem 3D* world are sprites. This includes all of the monsters like the assault troopers or octabrain, the goodies like medkits or holodukes, or the scenery items like bottles and barstools. In addition, there are several special sprites that help make the Build engine perform its many tricks. These special sprites will be discussed in chapter 6. This chapter will focus on the placement and selection of all types of sprites.

CREATING SPRITES

Sprites can be created in either 2D or 3D view mode. While in 2D view mode, place the mouse cursor on the spot where you would like the new sprite and press the S key. A new sprite will be created in that spot.

In 3D mode, you perform a similar action to place a sprite in a sector. Simply place the mouse cursor somewhere on the map and press the S key. Note that if the mouse cursor happens to be over a wall, the new sprite will be placed flat against the wall. If you're placing a more standard sprite like a monster or item for the player to pick up, place the mouse cursor on the floor of a sector before pressing the S key.

MOVING SPRITES

You can move the sprite around the map while you are in 2D view mode. To do so, just place the mouse cursor directly on the sprite, press and hold the left mouse button, and move the mouse. The sprite will follow the mouse cursor around until you release the mouse button. This moves a sprite along the x-y plane. To move a sprite up or down, you must be in 3D view mode. You select a sprite in 3D mode the same way that you select a wall, floor, or ceiling. That is, select the sprite by placing the mouse cursor on the sprite and holding down the left mouse button to lock that object as the current object being edited. Once the sprite is selected, press the PgUp and PgDn keys to move the sprite up and down, respectively, within the sector.

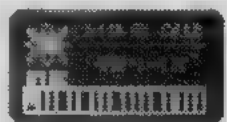
Note that it's possible to actually move a sprite below the level of the floor or above the level of the ceiling, which would obscure part or all of the sprite from view. You can align a sprite to rest directly on the floor of the current sector by using Ctrl + PgDn. Similarly, pressing Ctrl + PgUp aligns a sprite against the ceiling of the current sector.

APPLYING TEXTURES TO SPRITES

The *texture* that you give a sprite determines what that sprite will be: a monster, a key-card, a scenery item, or whatever you choose. You select a texture for a sprite in exactly the same way that you do for walls, ceilings, and floors. Simply press the V key in 3D mode while the sprite is selected with the mouse. Pressing the V key once will bring up a screen of all the sprite textures used so far, sorted by the number of times each one has been used on the map. Pressing the V key a second time brings up the list of all available textures.

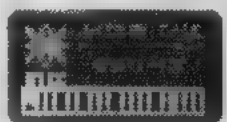
Note that on this second screen, both wall *and* sprite textures appear. There is nothing preventing you from using a wall texture for a sprite; in fact, this is exactly what you do to create some special objects like wall posters or magazines that appear to lie on the floor. Select the texture for the sprite on the selection screen by using the arrow keys and the PgUp and PgDn keys. Once you find the texture you want for the sprite, press the Enter key and the sprite will take on that texture.

SELECTING MONSTER TEXTURES



NOTE

You will find that some of the texture names in Build reflect earlier versions of monsters' names that were revised prior to the release of *Duke Nukem 3D*. In this case the LIZTROOP group represents the alien assault troopers that you frequently encounter during play.



NOTE

It is possible to select a monster texture that isn't named, because some textures are part of a set or sequence that is applied when the first *named* texture or frame is selected for the sprite. For example, if you apply a named texture showing a monster running, there very likely is an associated sequence of textures that show the same monster in different running poses so the monster is fully animated during game play.

When you are selecting a monster texture for a sprite you have placed on the map, notice that there are several textures for each monster. These textures are used to display all of the animation frames of each monster when it is running, shooting, dying, and so on. Because of this, each monster texture can be thought of as a *frame* for that monster. Also notice that some of the frames of each monster are named in the lower-right corner of the texture selection screen. For example, texture #1680, shown in Figure 5.1, is named LIZTROOP, texture #1681 is named LIZTROOPRUNNING, and texture #1682 is named LIZTROOPSTAYPUT. If you scroll through all of the textures for this monster, you will also see LIZTROOPDUCKING, LIZTROOPJETPACK, and LIZTROOPONTOILET.

When you place a monster texture on a sprite, be sure to use one of these named textures when you select it. These named textures work in conjunction with the GAME.CON file to define the way that the monster sprites will act. If you select a texture that isn't named, the monster won't act intelligently in the game; it will be inanimate.

Most of the monsters have a named frame that contains the word STAYPUT. When a monster sprite is placed on the map using this texture, that monster will remain in the sector



FIGURE 5.1: Select a texture to create a monster on the map. Notice the sprite name in the lower-right corner of the screen.

where he starts and will never leave that sector. Use this type of sprite for monsters that you want to guard a particular area, rather than pursue the player when they encounter one another.

SPRITE DISPLAY VIEWS

There are a variety of ways in which you can modify sprites to affect the way in which a player sees an object or monster during game play. In addition to selecting among three types of display modes for sprites, you can also change settings to affect a sprite's color and shading.

SETTING SPRITE DISPLAY MODES

There are three modes that Build can use to display a sprite. The first is a *pseudo-3D* mode that is used to display most monsters and objects. For objects, the sprite appears the same from all player viewpoints. This means that if you walk completely around the sprite, it will always face in your direction, as you can see in Figure 5.2. These types of sprites have no *side* or *back*. However, because monsters have multiple frames that show their side and back views, the effect is a much more believable one, as you can see in Figure 5.3.

The second display method flattens the sprite so it appears to lie flat against a surface. If you were to walk around a sprite with this type of display mode, you would see a definite *edge* at the side of the sprite. This display method is used mostly for sprites that are to be placed on a wall, like posters, calendars, and signs. The angle of this type of sprite determines which way the flat side of the sprite will face, as you can see in Figure 5.4.

The third display method also flattens the sprite but does so in the x-y plane, which allows the sprite to appear to lie on a floor or cling to a ceiling, as you can see in Figure 5.5. This sprite is used to great effect in The Red Light District's (E1L2) adult bookstore, in which it appears that magazines are lying on the floor. In the case of sprites

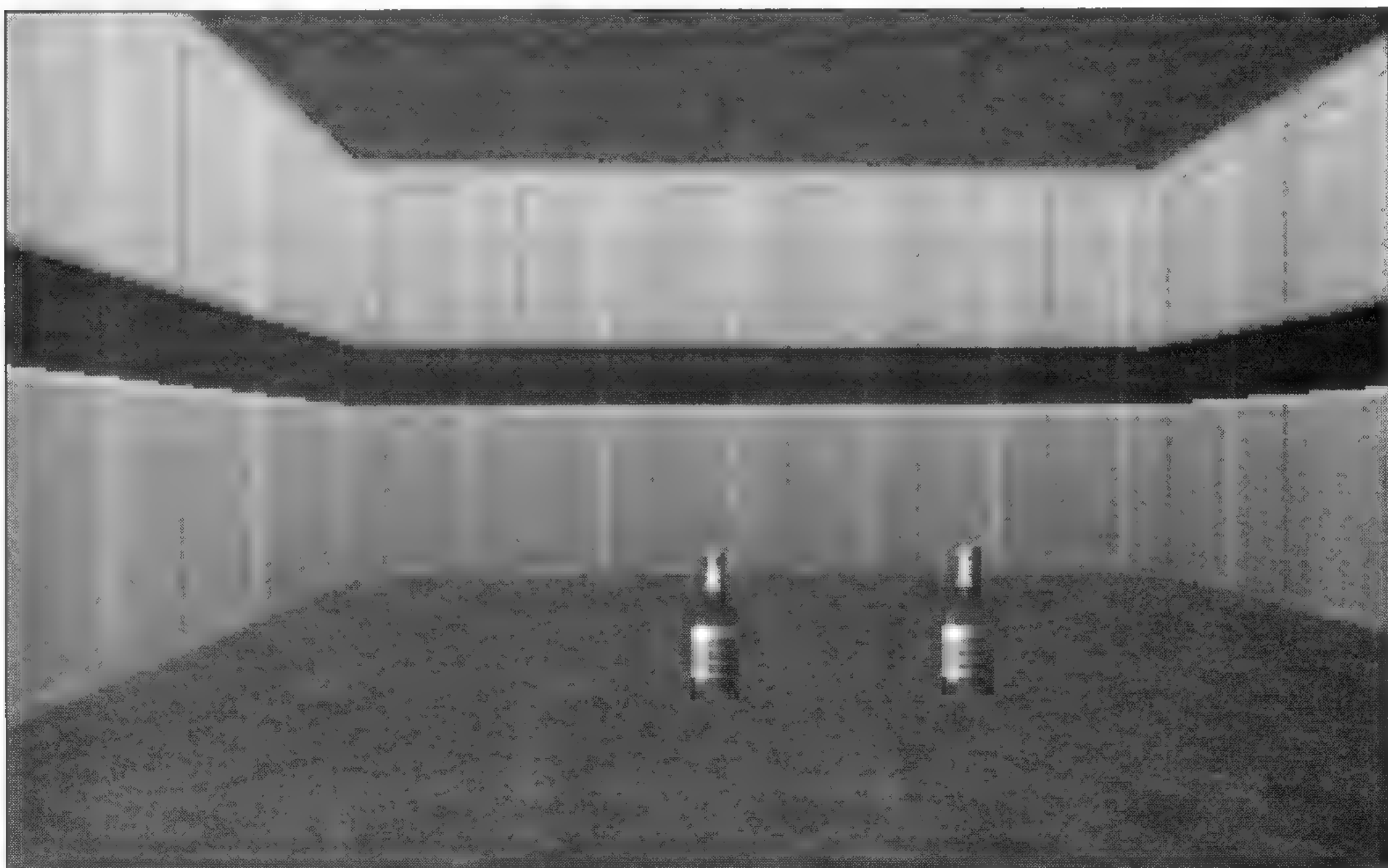


FIGURE 5.2: Although the angles of these two object sprites in pseudo-3D display mode were set in opposite directions, they still display the same from every viewpoint.



FIGURE 5.3: You can easily discern that these two monster sprites in the pseudo-3D display mode are facing opposite directions.



FIGURE 5.4: Here you can see four views of the same flattened sprite hanging in midair, at different angles.

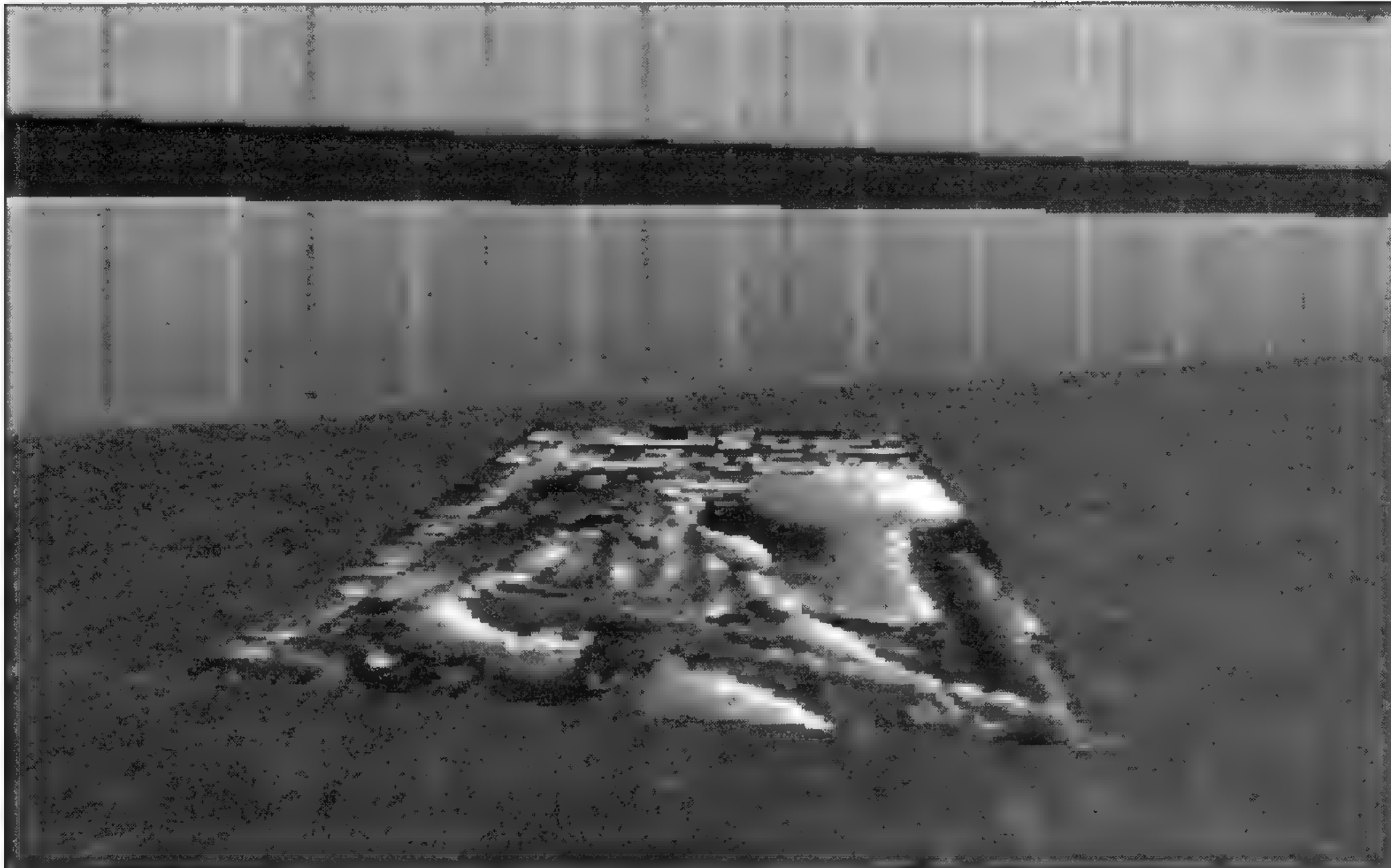


FIGURE 5.5: This sprite object is flattened so it appears to lie on the floor.

on a floor or ceiling, changing the angle of the sprite will rotate the sprite on the floor or ceiling so it faces a different direction.

To switch a sprite's display mode, use the R key with the mouse cursor on the sprite while in 3D mode. The sprite will toggle between the three display modes each time you press the R key. Make sure to hold down the left mouse button to lock the sprite as the object is being edited, because if you don't, changing the display mode will more than likely move the sprite so it is no longer under the mouse cursor.

SPRITE VIEWPOINT ANGLES

You have just seen that the angle a sprite faces is important for certain types of sprites, including monster sprites and sprites that have been flattened to lie against a wall or on a floor. For the objects that use the pseudo-3D display mode, angle is unimportant because the sprite will appear to face the viewer from every angle.

A sprite's angle can be determined in 2D mode by the *tail*, or straight line, that extends from each sprite. The tail points in the direction of the sprite's current angle. Figure 5.6 shows several sprites at different angles.

Changing a sprite's angle can be done in either 2D or 3D view modes by using the , (comma) and . (period) keys while the mouse cursor is selecting the sprite. Each time

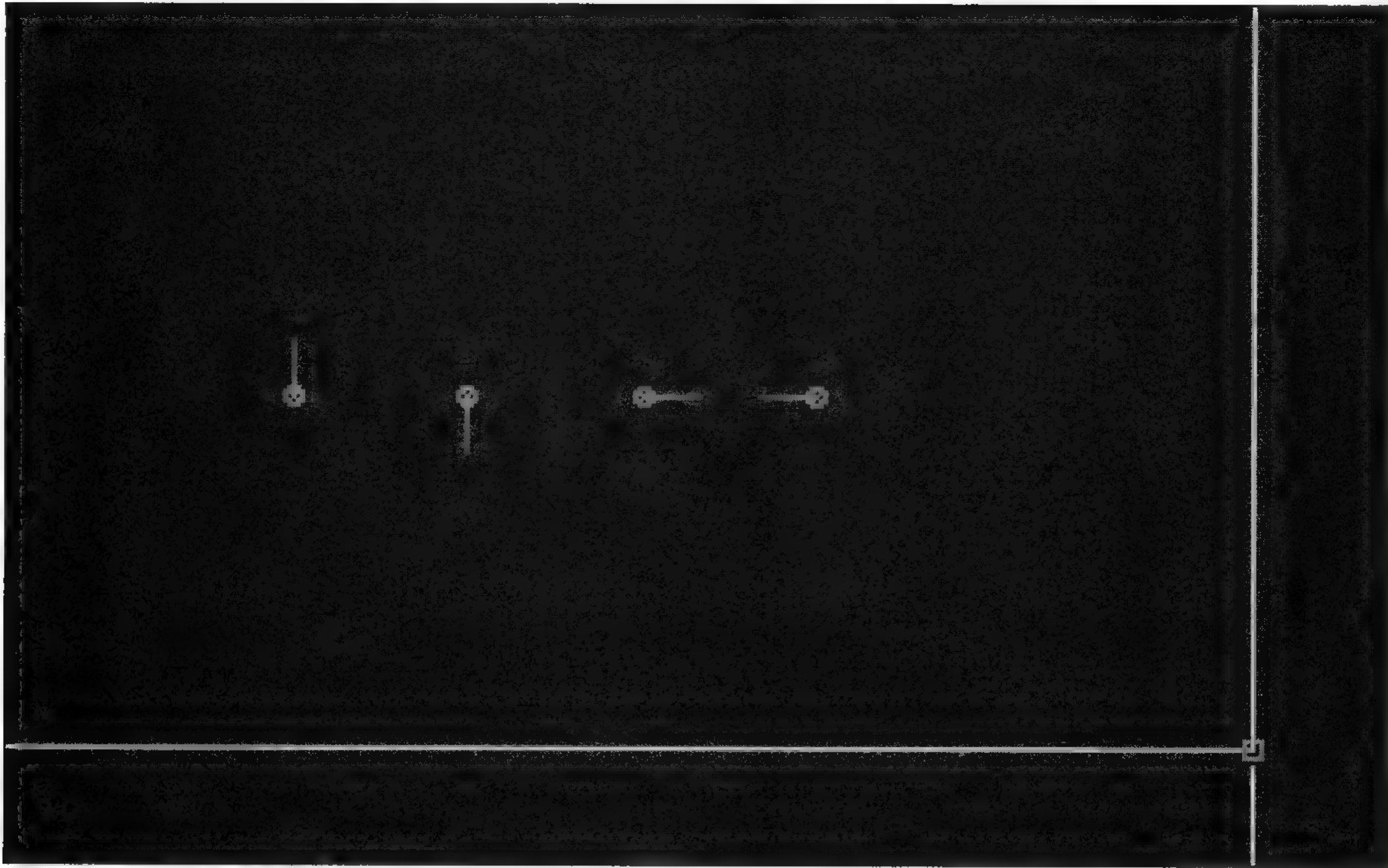


FIGURE 5.6: These four sprites are angled to face north, south, east, and west, respectively, from left to right, as indicated by the line or *tail* extending from the center of each sprite.

you press either of these keys, the sprite's angle will change by 22.5 degrees in the direction represented by the key. If you hold down Shift while using either key, you can fine-tune the sprite's angle by one degree at a time. One other key is useful in changing the angle of a sprite, particularly when you are trying to get a sprite against a wall. Press the O key in 3D view mode while the mouse cursor is on a sprite, and the sprite will toggle through the following three effects:

- ❖ The sprite will move to the nearest wall.
- ❖ The sprite's angle will be made perpendicular to the wall.
- ❖ The sprite's display mode will be made flat.

This technique is useful for making wall hangings and pictures. Note that the O key has already been discussed as the key used to change a wall's texture-painting orientation from a top-down to a bottom-up method, so make sure the mouse cursor is on the intended object when using this key.

SIZING SPRITES

Changing the size of a sprite, also known as *scaling* the sprite, is done in 3D view mode. To scale the sprite, place the mouse cursor on the sprite to be scaled, hold down the left mouse button to lock that sprite as the object being edited, and use the numeric keypad's 2, 4, 6, and 8 (arrow) keys. Holding down the numeric keypad's 5 key while using these keys will scale the sprite at a faster rate. To reset the sprite's size, press the / (slash) key.

Scaling sprites is very important because, for some strange reason, many sprites appear too large when you first select textures for them. Failing to scale these sprites down to size will cause them to be much too big for the objects they represent, as you can see in Figure 5.7.

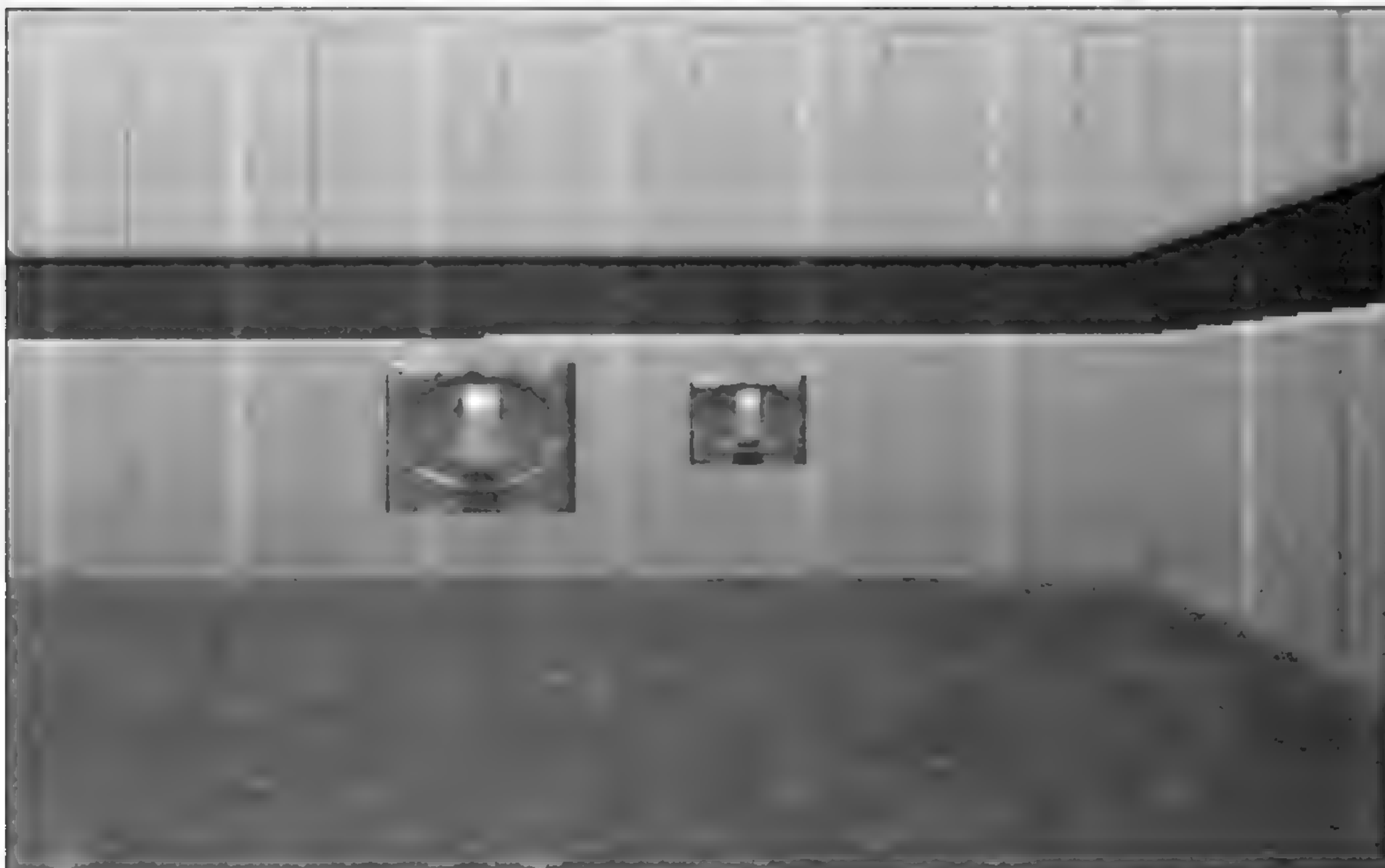


FIGURE 5.7: The Switch sprite on the left was much too big when it was first placed on the map. The one on the right has been sized to a more realistic scale.

SHADING SPRITES

If you leave a sprite at its default shade of 0, the sprite will take on the shade of the sector in which it resides. This makes for a natural effect, since a monster moving from a light sector to a dark one will change accordingly. However, you can override this behavior and change the shade of a sprite so it remains the same shade at all times. Shading

a sprite is done in 3D view mode using the numeric keypad's + (plus) and - (minus) keys while holding the mouse cursor on the sprite. You can also type a numeric value for a sprite's shade by using the ' + S (apostrophe + S) key combination in 3D mode.

One use of sprite shading is to lighten the shade of medkits and other power-ups that are in darker sectors so the player can see them more easily. These bright objects in dark sectors can also make effective traps, luring an unsuspecting player into a dark area for an ambush with whatever you desire.

MANIPULATING SPRITES

Similar to the ways in which you manipulate wall, ceiling, and floor textures, you can edit sprites that represent monsters or objects in a variety of ways, including copying them, deleting them, and flipping their textures. In addition, you can apply some special effects to sprites by controlling special bits for each sprite.

COPYING SPRITES

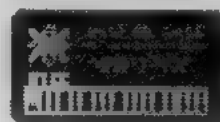
If you press the Tab key with the mouse cursor on a sprite in 3D view mode, the sprite will be placed into Build's clipboard. After this, each time you press S to add a new sprite in either 2D or 3D mode, an exact copy of this sprite, including the texture applied to it, will be placed on the map in this spot.

DELETING SPRITES

If you place a sprite on a map and later decide to delete it, you can do so in either 2D or 3D view mode by placing the mouse cursor over the sprite and pressing the Del key.

FLIPPING SPRITES

The texture of a sprite can be flipped in one of four directions by using the F key. Each



NOTE

Many users mistakenly use the Tab key to copy a sprite and then press Enter on the map, mimicking the keystrokes used to copy and paste wall textures. When you do this, the texture of the sprite will be applied as the texture for the current wall, floor, or ceiling, depending on the location of the mouse cursor. Note that the Tab key is still used for wall textures to copy a sprite, but the paste key is S.

time you press the F key, you toggle through the four orientations of the sprite's texture. Although flipping a texture so it appears upside down, as you see in Figure 5.8, is probably not useful for monster sprites, flipping some textures can increase the amount of sprites available to you.



FIGURE 5.8: This is a monster sprite with a flipped texture.

BLOCKING SPRITES

If you would like to make a sprite solid so a player cannot pass through it, press the B key when the mouse cursor is on the sprite in 3D view mode. The sprite will change from a cyan color to purple, which indicates that players cannot pass through it. A purple sprite is said to have its *Blocking bit* turned on. The term *Blocking bit* refers to the fact that the sprite will block a player or monster from passing through it. (For those of you with a computer programming background, the word *bit* refers to the fact that the blocking data is stored in a single bit.) Pressing the B key a second time will turn the Blocking bit off, allowing the player to pass through the sprite.

Another bit you can toggle for a sprite is the *HitScan bit*. You turn this bit on and off by pressing the H key in 3D view mode. The HitScan bit makes it possible for a

sprite to modify its appearance after it has been shot. You can also turn on the HitScan bit for vents and fans so they will appear altered when shot.

The Blocking and HitScan bits can also be toggled for sprites while in 2D view mode. The B key still toggles the Blocking bit as it does in 3D mode, but you need to press Ctrl + H to toggle the HitScan bit while in 2D mode. Because of this rather confusing series of keys, I recommend you do all the editing of Blocking and HitScan bits in 3D mode, where the keystrokes are similar.

CREATING TRANSLUCENT SPRITES

You can make any sprite partially translucent for an interesting effect. Use the T key with the mouse cursor on the sprite in 3D mode to make a sprite translucent. Repeated use of the T key will toggle the selected sprite between three degrees of translucence: 50 percent, 25 percent, and 0 percent (not translucent). This effect is often used for glass sprites or *ghosted* monsters. Figure 5.9 shows the same sprite using all three degrees of translucence. Although it is difficult to tell the difference between 25 percent and 50 percent translucence in Build, the effect is much more noticeable in the game itself.



FIGURE 5.9: This sprite appears from left to right with three different degrees of translucence: 0 percent, 25 percent, and 50 percent.

KEYSTROKE AND OPERATION SUMMARY

A summary of all of the keystroke sequences and operations covered in this chapter appears in Table 5.1.

TABLE 5.1 CHAPTER 5'S KEYSTROKES AND OPERATIONS

| KEY SEQUENCE/ OPERATION | FUNCTION | MODE PERFORMED |
|------------------------------------|--|-------------------|
| S | Create a new sprite | 2D/3D |
| Drag mouse with left button | Move sprite position | 2D |
| PgUp or PgDn | Move sprite up or down in sector | 3D |
| Ctrl+PgUp or Ctrl+PgDn | Align sprite to ceiling or floor | 3D |
| R | Change sprite display mode | 3D |
| , (comma) and . (period) | Change sprite angle | 2D/3D |
| O | Move and change sprite to lie flat against nearest wall | 3D |
| 2, 4, 6, and 8 (numeric keypad) | Scale sprite | 3D |
| + (plus), - (minus), or '+S | Shade sprite | 3D |
| Tab | Copy sprite to the clipboard (sprite in clipboard is sprite placed when inserted) | 3D |
| Del | Delete sprite | 2D/3D |
| F | Flip sprite texture | 3D |
| B | Turn on Blocking bit (make sprite unable to pass through) | 2D/3D |
| H (3D) and Ctrl+H (2D) | Turn on HitScan bit (allow sprite to be hit by projectiles) | 2D/3D |
| T (three modes) | Make sprite translucent | 3D |





Special Types of Sprites

The text file DEFS.CON that comes with *Duke Nukem 3D* defines the named sprites. If you open DEFS.CON with a text editor such as Windows Notepad, you will see a list of sprite names and numbers. The names are the same names that you see in Build on the texture selection screen or in 2D view mode as a label near the sprite.

Most of the sprites defined in the file DEFS.CON can be used by simply defining a sprite and setting the texture to the corresponding number in DEFS.CON. Some sprites, however, have some type of special function in the game and require a little more manipulation before they perform this special function. These special sprites are listed and described in this chapter. Each sprite is listed with its sprite number (the texture number), a description of its special function, and any additional values that must be set for the sprite to work properly. In many cases, these special sprites will be covered in even more detail in chapter 7, as they help to define the sector effects (like earthquakes) described there.

CREATING EFFECTS WITH LOTAG AND HITAG VALUES

You have already learned how to create a sprite and change its texture, shade, and palette. All of these attributes are set by placing the mouse cursor on the sprite and pressing the appropriate key sequence to change the desired value. However, there are two attributes that you can give sprites that haven't been discussed in this book: the *LoTag* and the *HiTag* of a sprite. These attributes are simply numeric values that help the

sprite perform some type of special function. Unfortunately, there is no more specific definition than that. Different types of sprites require different LoTag or HiTag values in order to perform their special function. In some cases, these two attributes help to link two or more sprites.

For example, consider the security cameras and monitors that can be seen throughout the *Duke Nukem 3D* levels. Both of these objects are created by placing sprites in different places on a map by the level designer. To make the two sprites work together, the LoTag and HiTag values for these two sprites have to be set in a particular way. There can be any number of cameras on a map, and any number of monitors. Proper setting of the LoTag and HiTag values will link certain cameras to certain monitors, to achieve the effect desired by the map designer.

LINKING SPRITES WITH UNIQUE VALUES

In many cases, the linking of two (or more) related sprites is done by making up a number and putting that number into the related sprites' LoTag or HiTag fields. For these types of sprites to work properly, the number you make up must be unique and used *only* for those sprites that are to be linked. For this reason, it's usually a good idea to keep a pen and paper handy to write down the unique LoTag and HiTag values and exactly where on the map you used each value. This way you don't reuse them later in the map.

Repeating a LoTag or HiTag value when you don't mean to can cause bizarre behavior in your level that can be a real bear to track down. For example, you could walk into a room expecting to hit a switch to start a sector rotating, but the sector is already rotating when you walk in. Why? More than likely, you tied the rotating sector's Activator sprite to a Switch sprite earlier in the level, by giving them the same HiTag value. Because there is no visual way in Build to easily see all the sprites linked in this manner, you'd have to hunt down the offending Switch sprite manually.

SETTING LOTAG AND HITAG VALUES

Now that you know the purpose of a sprite's LoTag and HiTag values, you need to know how to set them. You can set the LoTag or HiTag value of a sprite in either 2D or 3D view mode. In 2D mode, you place the mouse cursor near the sprite until it flashes, and press Alt + T to set the LoTag, or Alt + H to set the HiTag. In 3D mode, you also select the sprite by putting the mouse cursor on it, but you use the ' (apostrophe) key

instead of the Alt key; that is, press ' + T to set the LoTag value, and press ' + H to set the HiTag value.

It is useful at this juncture to know that both walls and sectors can be given LoTag and HiTag values. These settings are referenced in passing in the descriptions that follow, but you won't need to know how to set the LoTag or HiTag values for a wall or sector until the next chapter, when you will learn about special sector effects.

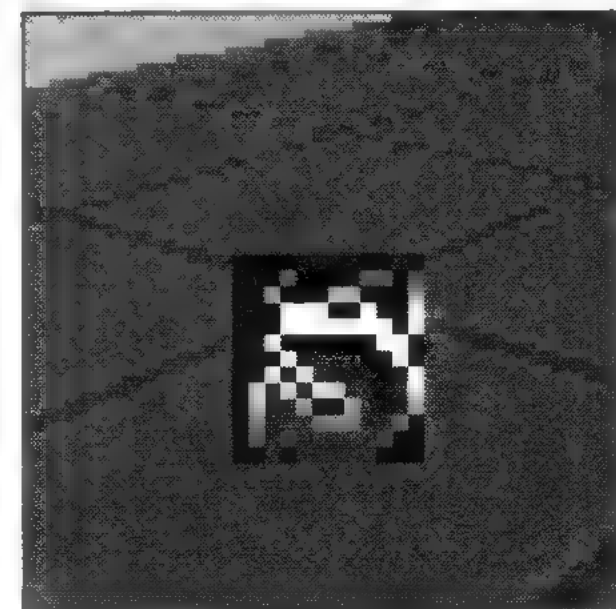
With these things in mind, read on for a list of all the special sprites and how they're created. Note that the number with each sprite name represents the sprite number, or the texture number. Also listed with each type of special sprite are the attributes that need to be set in order for that special sprite to perform its intended function.

The SectorEffector Sprite (I)

The SectorEffector sprite, also known as the SE sprite, is used to define many different types of special sector effects that can be performed by the Build engine. For example, a sector that teleports players and monsters is created by using a certain type of SectorEffector. A SectorEffector sprite is used by placing it in the sector that will perform the special effect. The SectorEffector sprite is automatically tied to the sector it resides in. There are many different types of SectorEffectors, and each is explained in great detail in chapter 7. Also, a map included in the Build directory in a file named `_SE.MAP` has a single room showing each SectorEffector in action.

The SectorEffector's LoTag value defines the *type* of SectorEffector. For example, when you see a reference to SE 15, this refers to a SectorEffector with a LoTag value of 15. By referring to the SectorEffector list in chapter 7, you would learn that an SE 15 is used to create what's known as a *slide door*. By contrast, the SE 14 sprite is used to define a *subway car*. This fact is *extremely* important to know for understanding the chapters that follow, and it is *not* well documented in the original Build documentation, so many people are originally puzzled by this. Just keep in mind that all SectorEffector sprites are created using texture #1, and the LoTag value of each individual SectorEffector that you place will control the function of that individual SectorEffector sprite.

By setting the same HiTag value for two or more SectorEffectors, you can cause several SEs to trigger at the same time. For example, you can create a room that will



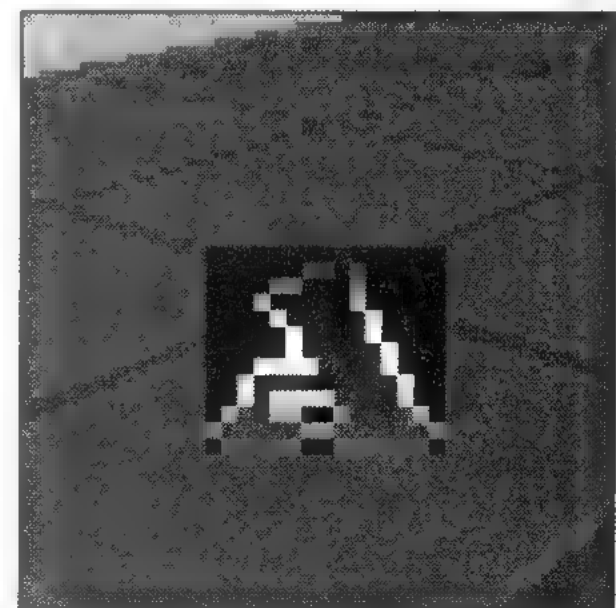
illuminate when a light switch is flipped. Because most rooms are in fact a group of individual sectors, you will need to tie each individual sector to the LightSwitch sprite by placing an SE sprite in each sector. To *link* all of these SE sprites, choose some unique number that is unused in your map to this point, and give all of the SectorEffector sprites that number as a HiTag value. Because the SectorEffector that acts as a light is SE 12, you would then give each of these SectorEffector sprites a LoTag value of 12.

You may also be required to set the palette, shade, and angle of each SectorEffector for it to perform its intended function. Once again, chapter 7 will provide a comprehensive reference to the settings required for the proper use of this sprite in all of its incarnations.

The Activator Sprite (2)

An Activator sprite is used to connect a Switch sprite or TouchPlate sprite (both will be described shortly) to a SectorEffector. Put a unique LoTag value in both the Activator sprite and the Switch (or TouchPlate) sprite. Any Sector-Effectors in the same sector as the Activator sprite will perform their function when the switch is hit.

For example, a ceiling can be easily made to rise by tying an Activator sprite to a Switch sprite. The Activator would lie in the same sector with the ceiling that will rise when the switch is flipped. A good example of an Activator sprite in action is the switch that starts the movie projector in Hollywood Holocaust, the first level in L.A. Meltdown (E1L1). This particular Activator sprite activates many different functions, including the lowering of the lights in the theater and the opening of the curtains. Assign the Activator sprite a unique LoTag value that matches a Switch or TouchPlate sprite's LoTag value, and set its HiTag value to 0.



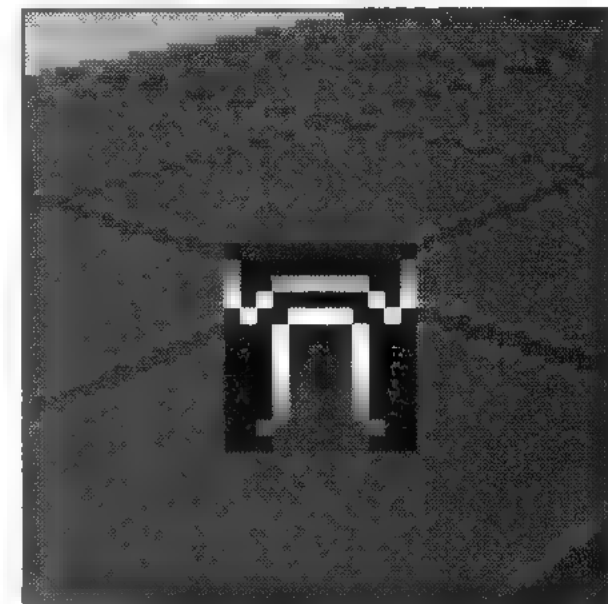
The TouchPlate Sprite (3)

A TouchPlate sprite is used to define a special sector that causes some type of action to occur when a player steps onto that sector. For example, an earthquake can be started when a player first walks into a room. To create this effect, place a TouchPlate sprite in

the sector you want to act as the trigger for the action. Give this sprite a unique LoTag value. Give at least one Activator or a MasterSwitch sprite this same LoTag value.

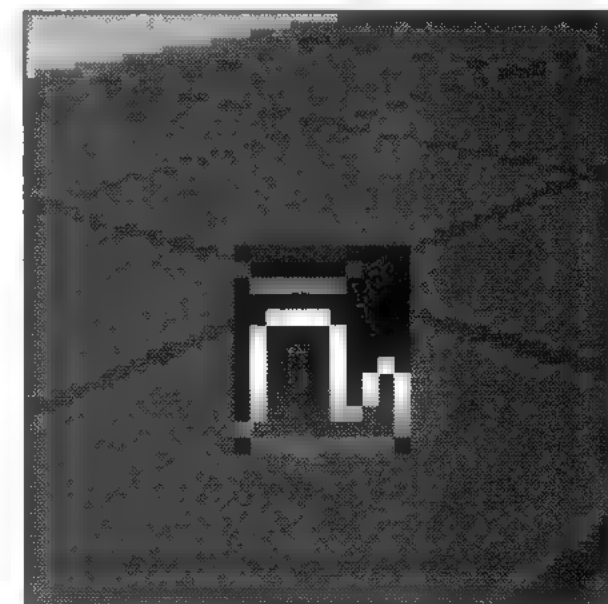
Note that you can tie the TouchPlate to as many Activators or MasterSwitch sprites as you want to have multiple actions occur at the same time. A good example of a TouchPlate sprite can be found as you leave the theater and head into the lobby in the Hollywood Holocaust level (E1L1). When entering the hallways to the north of the theater, an earthquake is triggered with a TouchPlate sprite.

If you set the TouchPlate sprite's HiTag value to 0, activation will occur every time the sector containing the sprite is entered. Setting the TouchPlate sprite's HiTag value to 1 or higher causes activation to occur as many times as the number you use, once each time the sector is stepped on. For example, if the HiTag value is set at 2, the action will happen the first two times the player steps on the sector, then it will not happen again.



The ActivatorLocked Sprite (4)

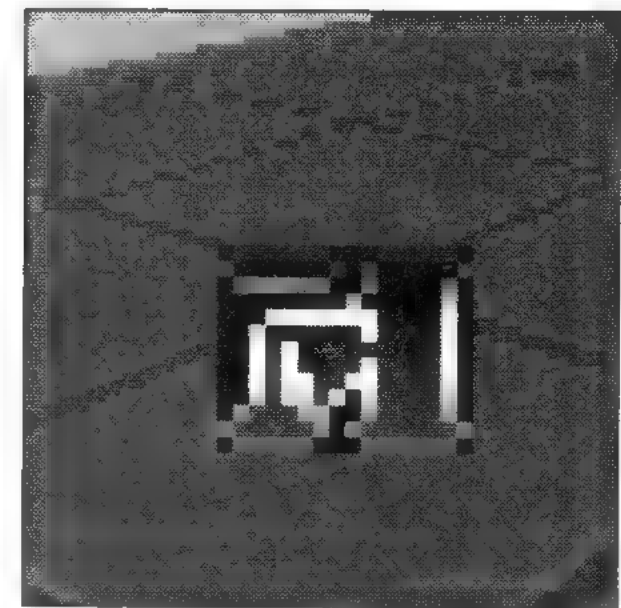
This sprite acts like an Activator sprite, but it will prevent the activation of the sector's LoTag function until a switch is pressed. The switch will toggle the function between a locked and unlocked state. This sprite is useful for locking a door until a switch (possibly a switch on the opposite side of the level) is flipped. An ActivatorLocked sprite can be found on several game levels, including in the adult bookstore on The Red Light District level (E1L2). The ActivatorLocked sprite prevents the red door near the cash register from being opened until the proper combination is selected by the player, who hits the red circular switches to the left of the door. The LoTag value for this sprite must match a Switch sprite's LoTag value; the HiTag value is set to 0.



The Music&SFX Sprite (5)

This sprite is used to play a sound. There are three different circumstances in which you can use this sprite.

- ❖ When a Music&SFX sprite is placed in a sector with a LoTag value, a sound will play when the sector's LoTag function is activated. For example, this sprite is placed in a door sector to define the sound the door makes when opened. In this case, you set the Music&SFX sprite's LoTag value to the sound number that is to be played. This sound is listed in the DEFS.CON file. Set its HiTag value to 0.
- ❖ When placed in a sector without a LoTag value, the Music&SFX sprite will play an ambient sound when the sector is entered. You can use this to set the mood for a certain place in your level. Note that not all sounds can be used as ambient sounds. Set this sprite's LoTag value to the sound number that is to be played, which is found in the DEFS.CON file.
- ❖ When a Music&SFX sprite is placed in a sector without a LoTag value, and the LoTag value of the sprite is greater than 1000, an echo effect will be heard in the sector. Use this echo effect in areas where you want to create the effect of a large space. Set its LoTag value to 1000 plus the amount of echo (0-255). Set the HiTag value to the maximum distance the sound can be heard.



SEE

See the section "Assign a Sound" in chapter 7 for a discussion on the use of the Music&SFX sprite in a sector effect.

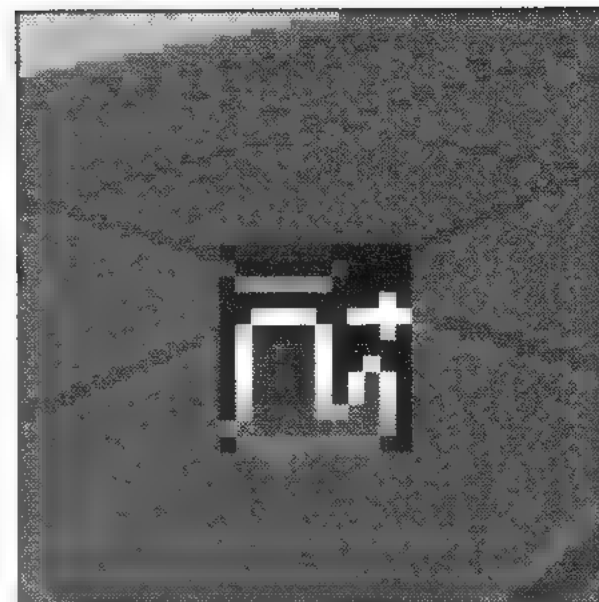


QUICK FIXES

For a complete list of the sounds available in the file DEFS.CON, see Appendix B.

The Locator Sprite (6)

Locator sprites are used in conjunction with the subway (SectorEffectors 6 and 14), the pig cop recon patrol vehicles (RPVs), or the two-way train (SectorEffector 30). In the case of the subway or RPVs, several Locator sprites are used to define a track that the object follows. In the case of the two-way train, two Locator sprites are used to define the two endpoints of the train.



SEE

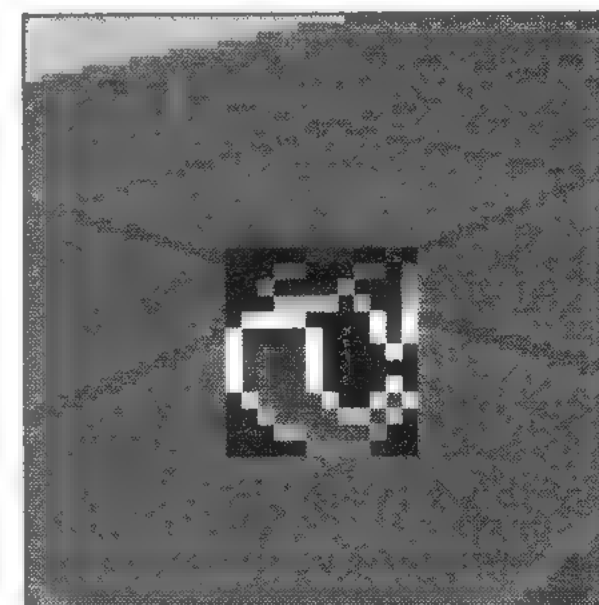
See the section "Two-Way Train (ST 31)" in chapter 7 for the proper use of the Locator sprite when creating these kinds of effects.

The Cyclor Sprite (7)

Placing a Cyclor sprite in a sector will make the entire sector pulsate in brightness. This sprite is good for creating special lighting effects on dance floors, in strip bars, and in bizarre alien areas. An entire group of these sprites is used to great effect at the very beginning of The Dark Side level in Lunar Apocalypse (E2L8). The

light in this area appears to come from red flashing lights on the ceiling and floor of the starting room.

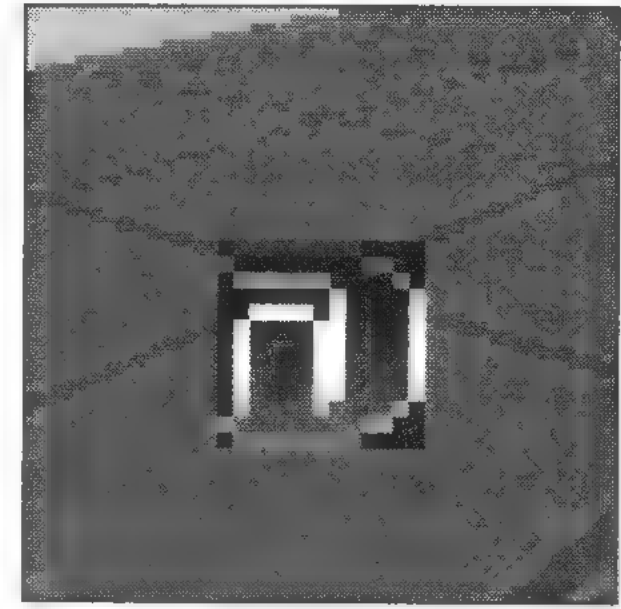
Set the LoTag value for this sprite to an offset (number) according to how bright the sector will start out. This attribute will rarely need to be set (only if you use several Cyclor sprites in succession, and their brightness relative to each other needs to be set). Set the HiTag value to 0. If a palette is set on the Cyclor sprite, the light in the sector will also pulsate with the color of this palette. Set the shade for this sprite to the highest shade value you want the sector to achieve. To set the darkest shade the sector will achieve, set the *sector's* shade value.



The MasterSwitch Sprite (8)

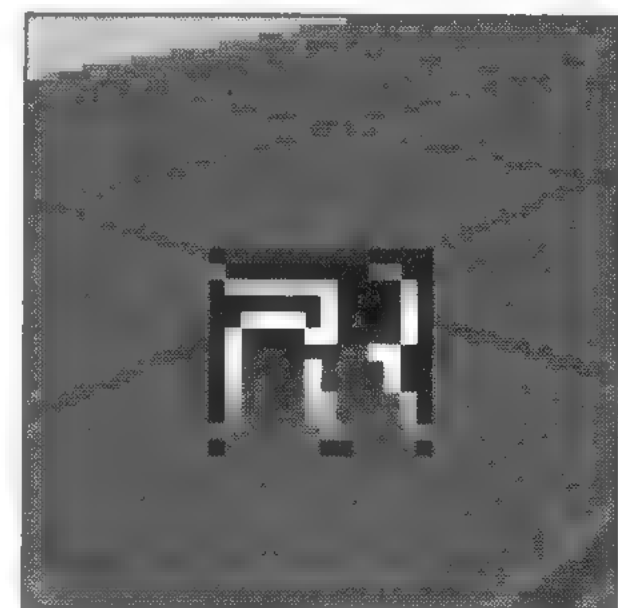
The MasterSwitch sprite will activate a sector LoTag or a SectorEffector in conjunction with a TouchPlate or a Switch sprite. This sprite basically works exactly like an Activator sprite, but the desired effect can be made to happen after a time delay.

You can set up a quick example of a MasterSwitch sprite by giving a long thin sector a LoTag value of 20, which makes the sector a door. In some nearby sector, add a TouchPlate sprite (3). Give this sprite a unique LoTag value. Place a MasterSwitch sprite in the door sector. Give the MasterSwitch sprite the same LoTag value as the TouchPlate's, and a HiTag value of 32. The HiTag value of the MasterSwitch sprite represents the time delay that occurs before the sector effect takes place, in this case the closing of the door. When you place this level in the game, the door will close a short time after you step on the sector with a touchplate.



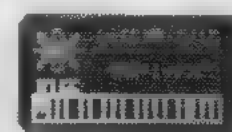
The Respawn Sprite (9)

This sprite will create a new sprite in conjunction with a TouchPlate sprite, according to the Build documentation. For the Respawn sprite, set the LoTag value so it is equal to the TouchPlate, MultiSwitch, or Switch sprite's LoTag value. Set the Respawn sprite's HiTag value to the sprite number for the character that is to spawn. The character must be represented by a *spawnable* sprite. Most sprites with names in DEFS.CON (like ammo, monsters, etc.) are spawnable.



QUICK FIXES

The Build documentation states that a MasterSwitch sprite can be activated only by a TouchPlate sprite, but this does not appear to be so. In the file `_SE.MAP`, the example of SectorEffector 36 (shrink ray shooter) has a MasterSwitch sprite that is activated by a normal Switch sprite.

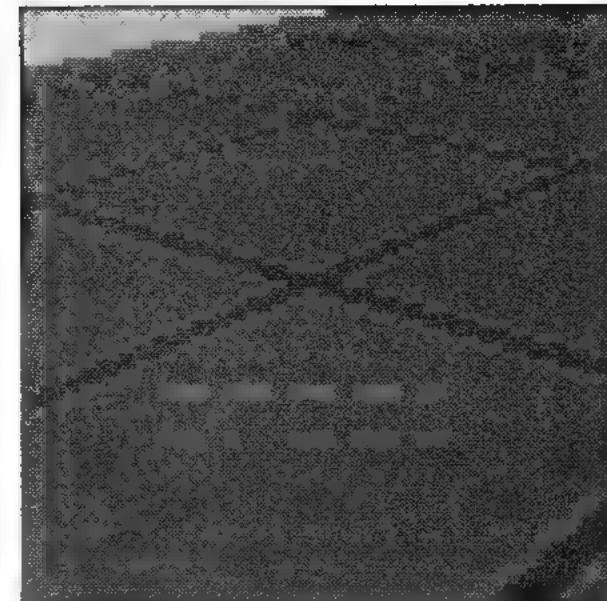


NOTE

I have read at least one online report of someone using the Respawn sprite successfully with a MultiSwitch sprite, and I have successfully used a Respawn sprite with a standard Switch sprite as well, despite the original documentation's claim that these sprites are activated only by TouchPlate sprites.

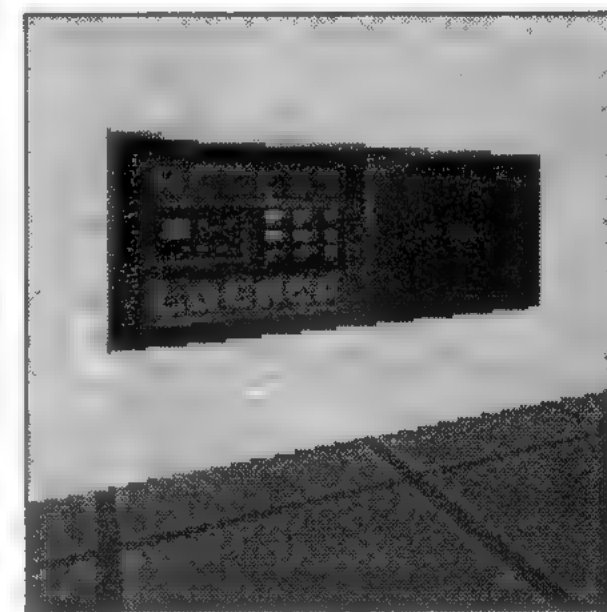
The GPSpeed Sprite (10)

This sprite is used in conjunction with many different sector effects. It usually defines the speed or distance of movement for the effect, depending on the values used for its LoTag and HiTag fields as well as the special sector types with which it works.



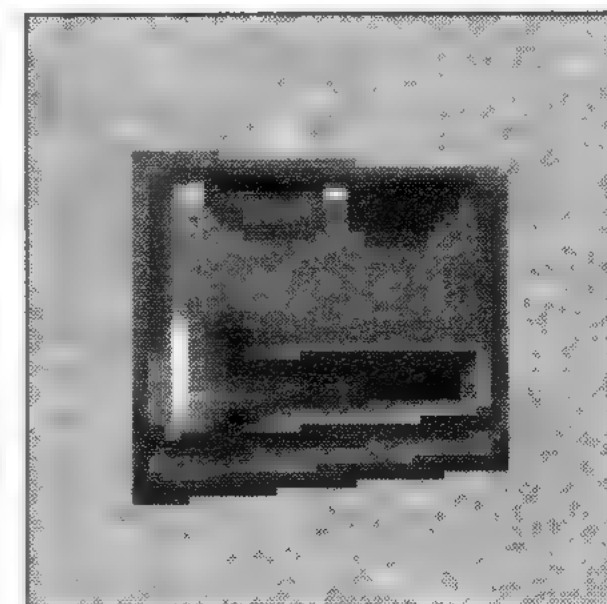
The AccessSwitch Sprite (130)

An AccessSwitch sprite serves as a receptacle for a keycard. This sprite is connected to an Activator or an ActivatorLocked sprite through its LoTag value. When the proper keycard is inserted, the Activator or ActivatorLocked sprite is triggered. Use the sprite's palette to define the required keycard. This sprite is first seen in the arcade of the Hollywood Holocaust level (E1L1). It opens the locked door that eventually leads to the end of that level. The HiTag value for this sprite is usually set to a sound number corresponding to a sound that is played when the Activator or ActivatorLocked sprite is triggered. For example, activation makes the little “plink” noise of a door unlocking. Set the palette for this sprite to a value that represents the correct keycard needed to trigger the Activator: 0 for the blue keycard, 21 for the red keycard, or 23 for the yellow keycard.



The Switch Sprite

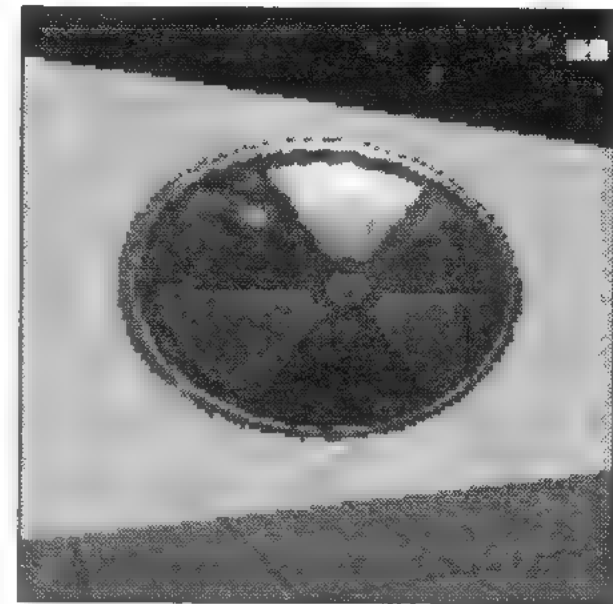
There are several Switch sprites, and the most common is #132. Switch sprites are used together with an Activator or an ActivatorLocked sprite to cause an effect to happen when the switch is hit. The switch can be defined to only appear in multiplayer games by setting the palette of the sprite. Note that one switch can cause many different effects at once, as in Hollywood Holocaust (E1L1), in which the switch in the projection room causes lights to dim, a curtain to open, and a movie to start. The LoTag value is set so it is equal to an associated LoTag value for an Activator or ActivatorLocked sprite. The HiTag value is set to a number representing a sound that plays on



activation. The palette for the sprite can be set to 0 for normal play or 1 for multiplayer games only.

The NukeButton Sprite (142)

This is the big red and silver button that a player hits to end a level. Set its LoTag value to 32767 to end a level normally when activated; set it to a number between 1 and 11 to exit to a bonus level. The HiTag value for this sprite is set to 0, and its palette can be set to either 0 for no special action or 14 for an exit to a bonus level.



The MultiSwitch Sprite (146)

This sprite representing a four-position switch activates four different Activator sprites. It is first seen in the Death Row level in L.A. Meltdown (E1L3) to control the opening and closing of the cell doors. To use this sprite, you will need four unique and consecutive LoTag values, such as x , $x + 1$, $x + 2$, and $x + 3$, where x equals any number you choose. Two of the four Activator sprites are activated each time the switch is flipped. If the Activators are named A, B, C, and D, this is the order they will be activated:

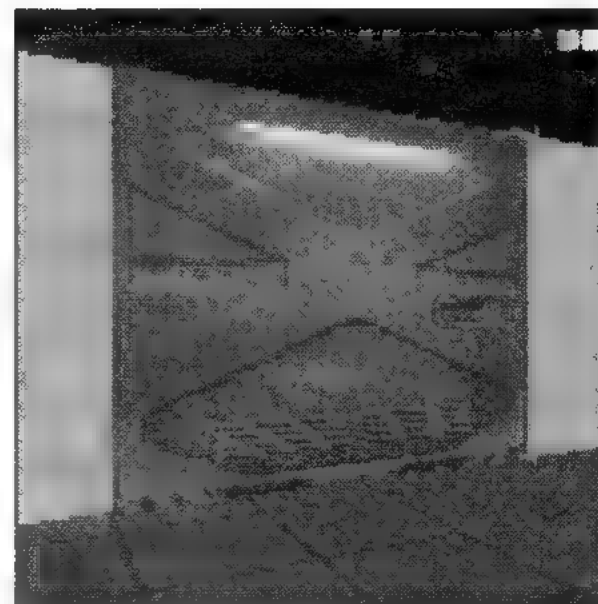


1. A and D
2. B and A
3. C and B
4. D and C
5. A and D

In Death Row, this causes one door to go up and one to go down each time the switch is hit. This rather specialized sprite is probably useful only in specific circumstances, although I'm sure some imaginative level designer will find a use for this function that I never thought of. The HiTag value for this sprite is set to 0.

The DoorTile# Sprite

There are several of these sprites—labeled DoorTile1, DoorTile2, and so on—with #150 being the most common selection. These sprites are unusual in that they are used as wall textures as opposed to being used as normal sprites. The special DoorTile texture will trigger Activator sprites when the user presses on the wall. This means that the opening of a normal door could also be made to create another action, like a crushing ceiling or an ambush from behind. For this sprite you set the wall



LoTag value so it is equal to the LoTag value or values of the one or more Activator sprites. Set the wall HiTag value to 0.

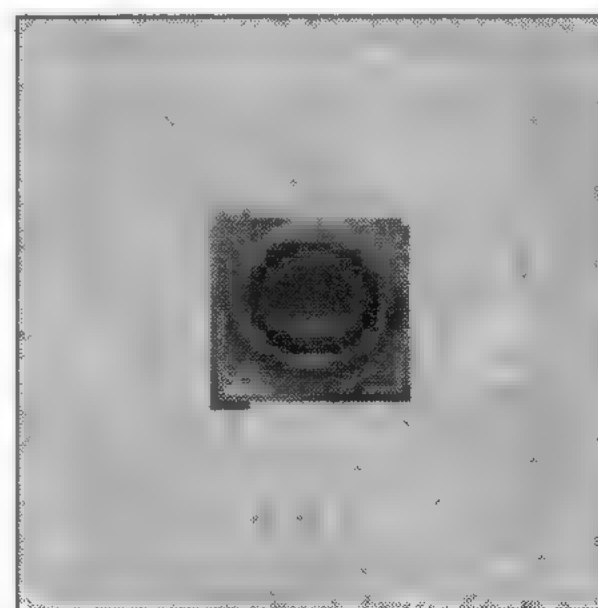


TIP

Note that the LoTag value goes on the wall that the DoorTile sprite is on, not on the DoorTile sprite itself.

on-off configuration to trigger an Activator or an ActivatorLocked sprite. The HiTag value is used to determine if the switch should be on or off to trigger the Activator sprite. One of the first times these switches are seen is in the construction office on The Red Light District level (E1L2). The proper combination reveals the switch that destroys the building across the street from the office. For this sprite you need to set the same LoTag value for all the DipSwitch sprites that are

used together. Also set the LoTag value for an Activator or an ActivatorLocked sprite to this same value. The HiTag value must be set to 0, if the switch must be off to trigger the Activator sprite, or to 1, if the switch must be on to trigger it.

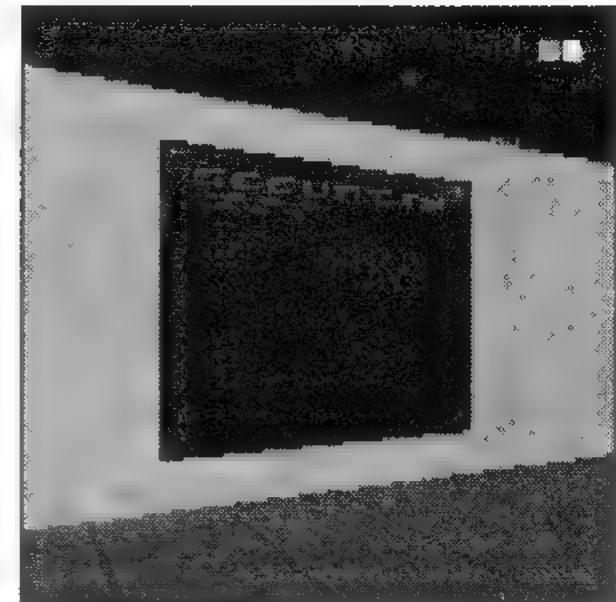


TIP

The DipSwitch sprites are not necessarily required to be placed together on a level. A group of scattered DipSwitch sprites that opens a huge cache of weapons or gives access to the secret level is a good way to keep the player guessing.

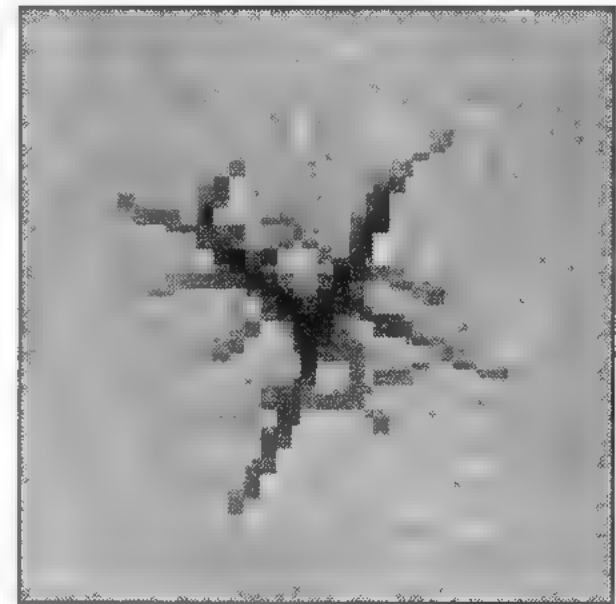
The ViewScreen Sprite (502)

This sprite creates the effect of the security monitors that give access to the camera views throughout the map. Set the LoTag value to 0. The HiTag value that you choose will link this ViewScreen sprite to all Camera sprites with the same value in their LoTag fields. That is, a ViewScreen sprite with a HiTag value of x will control all Camera sprites with the same LoTag value of x.



The Crack1-Crack4 Sprites (546-549)

The Crack sprites that you can place on a wall activate explosions when they are hit with an explosive device (an RPG or a pipe bomb). The LoTag value for this sprite is set to 0; the HiTag value should be equal to all the HiTag values of the SE 13s (C-9 explosive) that are to activate.



SEE

See the section "C-9 Explosive (SE13)" in chapter 7 for instructions on how to set up an explosion using a Crack sprite.



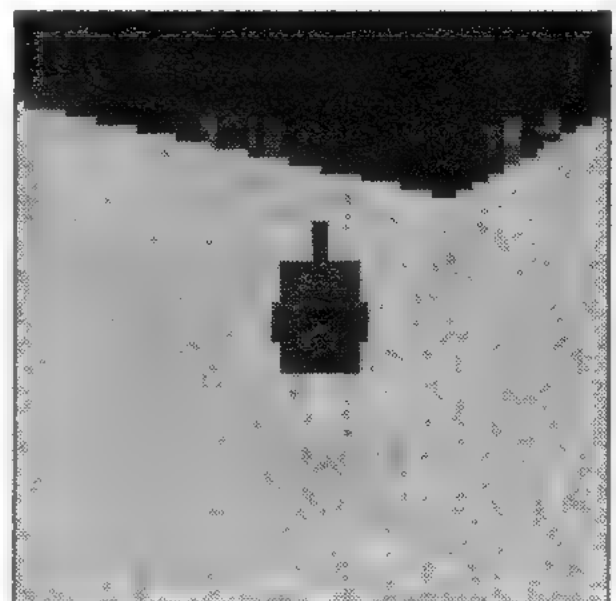
QUICK FIXES

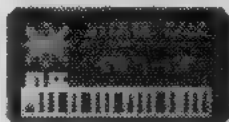
The original Build documentation for the attribute settings of the Camera1 sprite are incorrect. The LoTag value should be set to the ViewScreen sprite's HiTag value. The HiTag value should be set to half the turning radius of the camera.

The Cameral Sprite (621)

This sprite represents the cameras that connect to the ViewScreen sprites. Its HiTag value represents half the turning radius of the camera.

To set the turning radius in Duke Nukem angles, 512 equals 90 degrees, so set the HiTag value to 256 if you want the cameras to pan 90 degrees (the 256 represents 45 degrees in each direction, for a total of 90 degrees). The shade should be set to a value for the angle that the camera faces to the ground (used for elevated sprites).





NOTE

You can flip the sprite to make the camera look like it's mounted from a pole instead of from the ceiling, but do not make the camera rotate if you wish to use this type of pole-mounted camera. Only the first frame of the camera animation will flip, so an animated sprite will not look correct when flipped.

The angle setting should equal the initial direction the camera faces.

The CanWithSomething Sprite (1232)

This is a garbage can that spawns a sprite when it is destroyed, making it appear that the can contained an item. The LoTag value corresponds to the sprite that's



spawned, so the correct value must be set to the

sprite number for the object that will spawn, which must be spawnable. Set the HiTag value to 0.

The SeeNine Sprite (1247)

Also known as a C-9 sprite, this yellow canister explodes when shot. SeeNine sprites are often used in groups to blow holes open in walls. They can also be used by themselves, where they make great traps for players in either single-play or DukeMatch mode. The LoTag value represents a time delay until the explosion occurs. The HiTag value is set equal to the SE 13's (C-9 Explosive Sector-Effector) HiTag value, which causes activation.

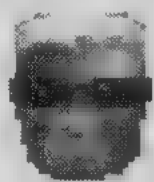
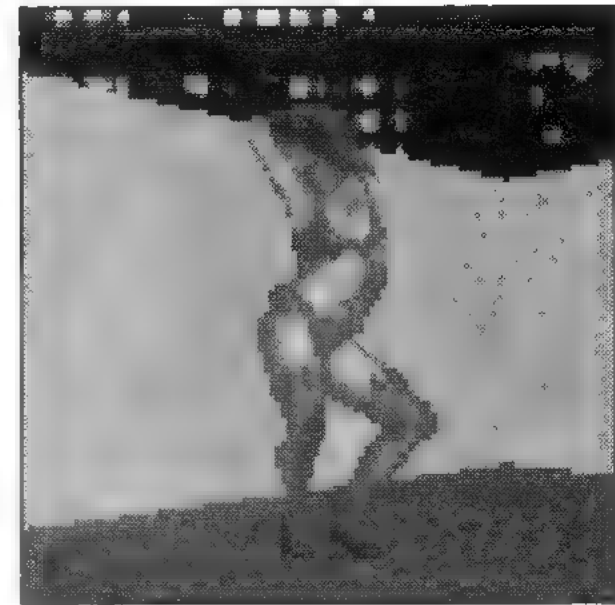


SEE

See the section "C-9 Explosive (SE 13)" in chapter 7 for instructions on how to create an exploding hole in a wall.

The Fem# Sprite (1312)

These sprites—labeled Fem1, Fem2, and so on—represent the dancers and other women in the game. They will activate a Respawn sprite to teleport in a monster when a woman is killed. The LoTag value is set to 0, and the HiTag value is set equal to the LoTag value of the sprite that respawns upon activation.



QUICK FIXES

The Build documentation is very incorrect in its description of the Fem# sprites. It refers to the *FemPic* series of sprites, which are the women posters seen throughout the maps, and which don't seem to have any type of special effect.

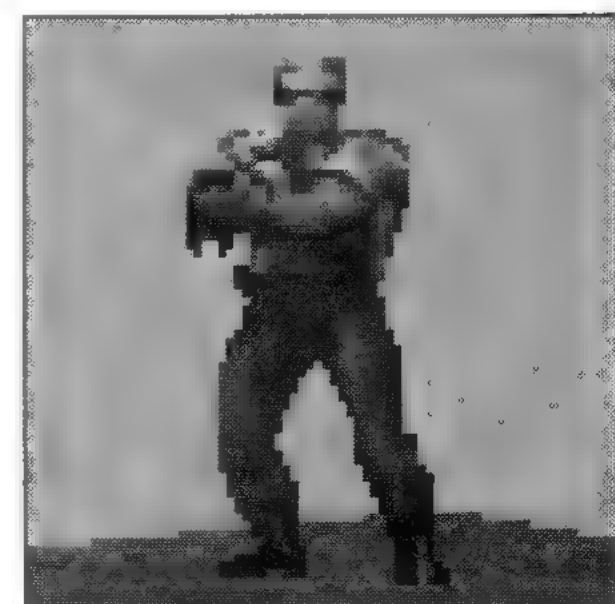


SEE

See chapter 10 for ideas on creating memorable levels for play involving Aplayer sprites.

The Aplayer Sprite (1405)

This sprite is Duke Nukem himself. You use the Aplayer sprite to define the Co-Op and DukeMatch starting points. Place seven of each type in every map. Put the Co-Op points near the actual player one start location (the brown arrow in 2D mode), and scatter the DukeMatch points all over, making sure to keep in mind player balance. The LoTag value is set to either 0 for a DukeMatch starting location or 1 for a Co-Op player starting location. The HiTag value is set to 0.



The use of these special sprites should become very clear and familiar to you as you learn to incorporate them into special effects, which you will do as you work through chapter 7.





Special Sector Effects

Now comes the hard part. You've learned how to make rooms out of sector groups; apply light, shade, and palette; and place all kinds of goodies in them. But your levels so far have been kind of static—not much movement going on. *Duke Nukem 3D*'s original levels have doors, elevators, conveyor belts, earthquakes, and walls exploding right in front of your face. To make these kinds of effects, you have to delve into the world of sector effects, which you create with Sector Tags, SectorEffectors, and other elements. By using Sector Tags and SectorEffectors to create sector effects, your levels will really begin to come to life.

So what makes creating these effects so complicated? For one thing, many of these effects require more than one step and more than one construct to create them. Second, and unfortunately, you cannot see these effects working in Build. To see your sectors in action, you have to exit Build and go into the game itself. This means making a little part of the level in Build, saving it, going into the game to try it out, going back into Build for editing, going back into the game for another round of play-testing, and so on. In this chapter, you will see how the different elements of a special effect are put together.

PREPARING TO DEFINE SPECIAL EFFECTS

It goes without saying that this chapter gets into some pretty complicated avenues of level design. The essence of building animation into your levels is understanding and building relationships between the special effects you want and the Sector Tags, SectorEffectors, and other elements that make those effects happen. If you'll recall, a *SectorEffector* is one of the special sprites you can place into your level. A sprite with texture #1 is a SectorEffector, and it looks like a stylized letter S when you see it in Build's 3D view mode. Although there is only one SectorEffector sprite, there are many different *types* of SectorEffectors, and each type is selected by assigning the SectorEffector a LoTag value.

Recall that I mentioned in passing in chapter 6 that, in addition to sprites, sectors and walls can also be given LoTag and HiTag values. The LoTag value of a sector usually defines the type of special effect that sector is going to perform, but this is not always the case. In fact, many different methods are required to define each type of special effect. Sometimes, simply assigning a sector a LoTag value sets up the desired effect. Other times, a sector's LoTag value is combined with a SectorEffector to produce the effect. Still other times, you will need a sector LoTag value, a SectorEffector, an Activator sprite (2), and a Switch sprite to produce the effect you desire.

The fact that each effect is produced differently is one reason why creating them can be so difficult. It is very hard to memorize how to do each type of effect. However, having a nice example of each type of effect described here should not only ease the initial learning curve of each effect, but also provide an excellent reference tool for you as you create your Build levels.

REVIEWING WHAT YOU KNOW

Before you jump right in, however, you should assure yourself that you're comfortable with everything you've done up to this point. Consider the concepts covered in previous chapters. By now, you should be able to do the following:

- ❖ Create sectors of any shape or size
- ❖ Connect two sectors, split a sector, and join two sectors into one
- ❖ Create a child sector inside a parent sector

- ❖ Raise sector ceiling and floor heights
- ❖ Place new sprites on the map and modify all of their attributes, especially the HiTag and LoTag values

If you don't feel comfortable doing all of these things, then I recommend that you go back through the previous chapters and review those concepts that are still a bit difficult for you. You may want to go back and practice the steps in the areas you are having difficulty with until you're comfortable with them. The exercises in this chapter focus more on procedures for performing the new, more complex, operations and assume that you can do all of the basic operations summarized previously. For example, the instructions you'll encounter in this chapter might stipulate that you create a child sector, without including the guided procedure for doing so.

As you work through the rest of this chapter, you'll notice that exercises won't include the selection of textures for the walls, floors, and ceilings. This chapter's focus is more toward the *mechanics* of making special effects work, as opposed to making them look *good*. Of course, you'll have to select textures for some of the effects appropriate to the rooms that will contain the effects you're creating. For example, doors will require a door texture, and conveyor belts will look more like conveyor belts after you choose the appropriate textures.

PRINTING A HANDY REFERENCE

You may find it very useful at this point to find the file DEFS.CON in the *Duke Nukem 3D* main directory (if you installed the game to the default directory, its name will be *Duke3d*) and print it. All of the sprites are defined by name in this file, as are all the sound effects. You will be choosing sprites and sounds for their various effects, and it helps to have this list available in hard copy as you work so that you won't have to exit Build to look one up on your computer. Since this file is a simple ASCII text file, you can either print it from a program such as Windows Notepad, or simply type **PRINT DEFS.CON** from the DOS prompt.

INTRODUCTION TO SECTOR EFFECTS

Special effects that occur in sectors during game play largely involve animation of Duke's physical environment, as the player *blazes* through the game's levels. If you've played *Duke Nukem 3D*, you know that doors open and close, platforms move in and out, elevators move up and down, cogs and other architectural structures rotate or spin, and rooms appear to shake from explosive impacts or earthquakes. These and many other special sector effects help make the game experience quite thrilling.

To help acquaint you with the complexities that go into developing these amazing effects, let's begin with an exercise that will demonstrate some of the more typical procedures that go into creating sector effects. You will create a door that simply opens and closes.

CREATING A SIMPLE *DOOM*-STYLE DOOR

If you've ever done any *DOOM* level editing, you'll recall that a door in *DOOM* is simply a long thin sector that has its ceiling lowered down to its floor. The act of opening the door actually raises the ceiling of this thin sector to match a neighbor ceiling. You can create *Duke Nukem 3D* doors that work the exact same way. You'll start by just getting a door to open and close.

Create Some Sectors

Begin by starting Build with a new map in 2D view mode. Draw a square sector, and next to it but not touching it, draw another square sector. On each of the two lines where the two sectors almost touch, press the Ins key two times to split these lines into three pieces. Your map should look like the one shown in Figure 7.1.

Now, make a new sector by placing the mouse cursor on each vertex you just created and pressing the spacebar. The result will be a smaller sector that joins the first two sectors. This smaller sector will be your door sector.

Assign a LoTag Value

Place the mouse cursor somewhere within the new *door* sector. Press the T key to edit the sector's LoTag value. By looking at the Build documentation, you will find that a normal ceiling-based door is Sector Tag 20. Type **20** and press Enter.

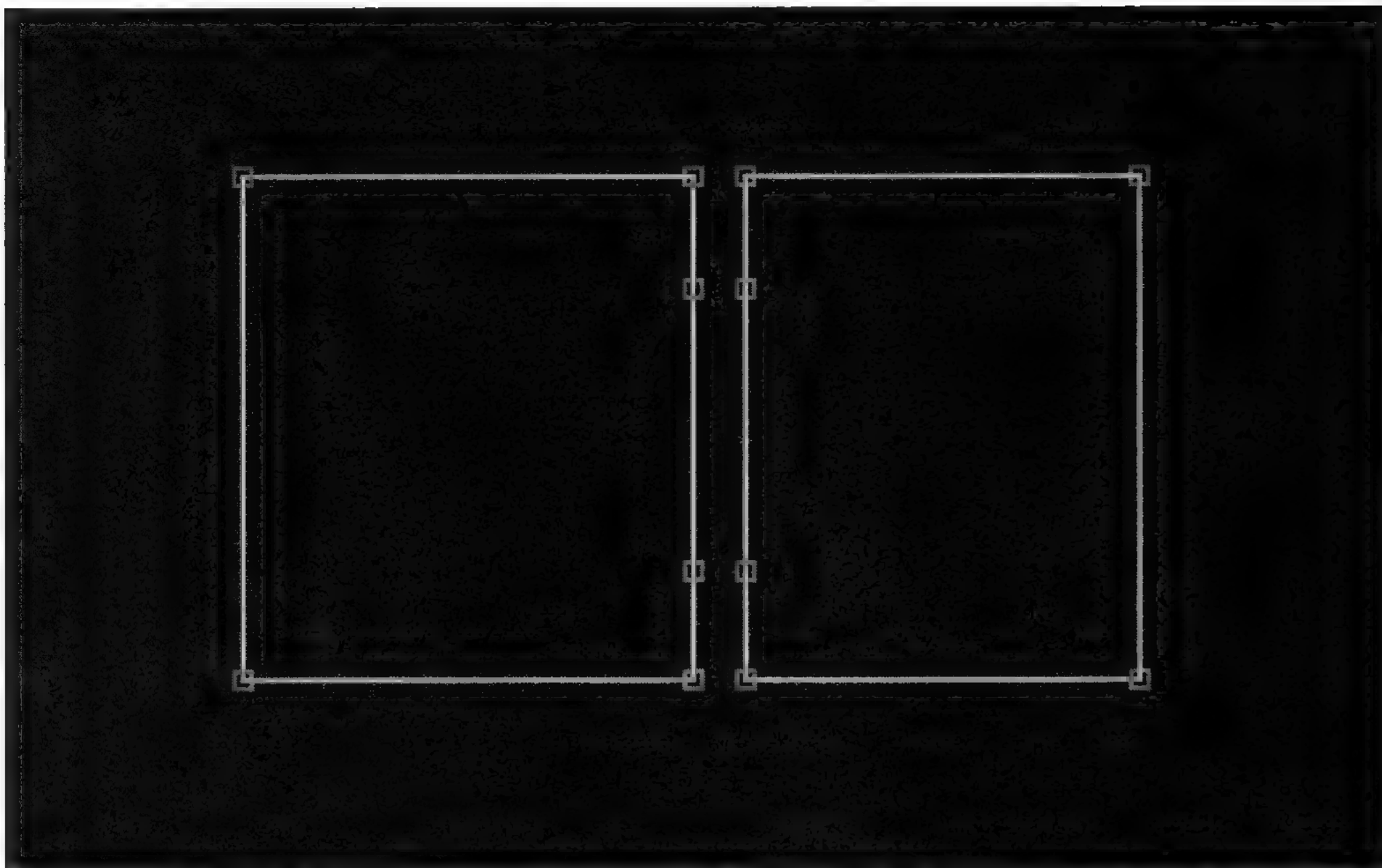


FIGURE 7.1: The left sector's east wall and the right sector's west wall are each split into three sections.



TIP

You can assign sector LoTag and HiTag values in either 2D or 3D view mode. In 2D view mode, place the mouse cursor inside the sector; press the T key (for the LoTag) or the H key (for the HiTag); and type the value. In 3D view mode, place the mouse cursor on the sector's floor or the ceiling; press the ' (apostrophe) key along with the T key ('+T) to set the LoTag value or along with the H key ('+H) to set the HiTag value; and type the value.

Try Out Your Door

Now, save your map, quit Build, and then start the game with the map and try out the door! The door you made will start off in the open position; walk up to the door's opening and press the open button (usually the spacebar on the keyboard). You will see the door close. Try opening and closing the door a few times, and then go back into Build because you're not done with this door yet. When you return to Build, start off in 2D view mode.

There are a few things you should note about your door.

1. The door didn't make any noise.
2. The side walls of the door moved along with the door. Most doors look better if these textures don't move.
3. The door was open. Most doors should be closed.
4. The door will never close on its own. You might want a door that automatically closes after a period of time.
5. The door will always open and close at the same speed. You might want a faster or slower door.

You can correct each of these characteristics in turn.

Assign a Sound

There is a special sprite called the Music&SFX sprite (5) that you can use to assign sounds to a sector effect. This sprite, when used in conjunction with doors, helps define the sounds that the door will make when it opens and closes. For this sprite to work correctly with a door, it must be *inside* the door's sector. Putting the sprite on the boundary of the door sector won't work. This is where deft use of Build's *grid* functions becomes really important. Press the G key to make the grid squares small enough that some of the grid lines cross inside your door sector. Alternately, you can temporarily turn off the grid by pressing the L key.

Once you have the grid configured so you can place a sprite inside the sector, place the mouse cursor somewhere inside the sector, and press the S key to insert a sprite. Now you need to turn this sprite into a Music&SFX sprite. To do this, switch to 3D view mode, find the new sprite in the door sector, and put the mouse cursor on it. Then press the V key twice to bring up the list of all the available textures. The sprite you're looking for is number 5; it looks like a stylized letter "M," and it is in the first row of all the available sprites. Select this sprite and press the Enter key.

Now you need to decide what sound you want to be played. This is where having a printed copy of DEFS.CON comes in handy. Find a sound that looks like one associated with a door opening. Sound 74, for example is named DOOR_OPERATE1. To specify which sound the Music&SFX will play, you need to change the LoTag value of the Music&SFX sprite to the same number associated with the sound. Put the mouse cursor on the Music&SFX sprite, and press ' + T (apostrophe + T) to change the sprite's LoTag value. Type the number 74 to use the sound DOOR_OPERATE1, and press Enter.

The sprite should now have the label “0,74 MUSICANDSFX” when you see it in 2D view mode.

Stop the Side Walls from Moving with the Door

Curtailing the movement of the door sector’s side walls is just a matter of orienting the textures on these walls to the bottom of the wall. You learned in chapter 4 that wall textures are painted in a top-to-bottom direction by default. To switch the drawing method to bottom-to-top, go into 3D mode, put the cursor on one of the side walls of the door, and press the O key. You may or may not see the texture on the wall move: Don’t worry if you did not, this just means your current texture looks exactly the same, whether it’s drawn from the top or bottom on this particular wall height. Do the same for the other side wall of the door sector.

Close the Door

Making the door so that it starts off in the closed position is a simple matter of bringing the door sector’s ceiling down to the floor level. Go into 3D mode to accomplish this. Place the mouse cursor so that it’s directly on the thin ceiling texture of the door sector. Press the PgDn key, and the door should close a bit. (It may take a few tries to get the door sector selected, especially if all the adjoining ceilings currently all have the same texture.) Once you have the correct sector, hold down the left mouse button and press PgDn until the door sector’s ceiling won’t go any farther. Note that you could also leave the door half-closed, if it suited you to create a partially opened door for your level.

Make the Door Close Automatically

You can use SectorEffector 10 to make an automatic-closing door. To use this sprite, switch to 2D view mode. Place a sprite into your door sector by moving the mouse

cursor inside it and pressing the S key. (Make sure the sprite is completely inside the sector, not on one of the edges.) Change the sprite to Sprite 1, which is the SectorEffector sprite. You do this by going into 3D mode, moving the mouse cursor onto the sprite, pressing the V key twice, and selecting texture #1, which looks like a stylized letter “S.” To make this



TIP

If you’d rather set SE10 in 3D view mode, place the mouse cursor on the sprite, press ‘+T (apostrophe+T), and type 10.

SectorEffector an SE 10, change the LoTag value of the sprite to a 10 by switching to 2D view mode, placing the mouse cursor on the sprite, pressing Alt + T, and typing 10.

Finally, you need to decide how long you would like the door to wait before it closes. The HiTag value of the new SE 10 sprite controls this delay. Place the mouse cursor on the new sprite, and press Alt + H to edit the HiTag's value in 2D view mode or ' + H (apostrophe + H) in 3D view mode. A value of 32 equals one second, so choose any value you want for the door to stay open before it automatically closes. Choosing a short time value like 16 (for a half second) is good for creating a *paranoia* effect for the player because the door starts closing only a half second after it is opened, usually before the player even has a chance to go through it. On the other hand, a value of 128 (four seconds) provides a good time for the player to open the door, glance at what might lie beyond, and step through it.

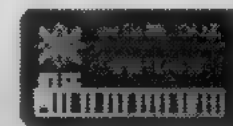
Change the Door's Speed

Changing the rate at which the door opens and closes is done with yet another special sprite—the GPSpeed sprite (10). Place one of these sprites inside the door sector just as you did the Music&SFX and SectorEffector sprites previously. To set the speed of the door, set the LoTag value of this sprite to the desired speed. A setting of 256 seems to be the default speed; 512 is pretty quick, while 64 is a nice slow ponderous door.

Try Out Your Door Again

Go back into the *Duke Nukem 3D* game, and try out your door again. It should now make a noise when opened and closed, the side walls should look more natural when the door moves, and the door should open faster or slower than it used to, depending on the speed you chose for the GPSpeed sprite. Finally, the door should close by itself after a few seconds. If any of these things do not happen, review the procedures above, and get the door to behave this way.

Before moving on, review the following steps that summarize how you created and modified your door effect.



NOTE

If you forget how to add a sprite and change its number, look at the steps for adding a Music&SFX sprite discussed previously, but make the sprite texture 10 instead of 5.

1. Create a thin sector bordering two existing sectors.
2. Make the sector's LoTag value 20 (for a ceiling door).
3. Orient the side walls' textures so they don't move (press the O key in 3D mode).
4. Add a Music&SFX sprite (5) for the sound.
5. Bring the door's ceiling down to the floor.
6. Add a GPSPeed sprite (10) to change the door's opening and closing speed.
7. Add a SE 10 sprite to make the door close automatically.

Some of these steps, of course, are optional, depending on how you want your door to act. For example, the default speed may be fine for your door, so you would skip adding the GPSPeed sprite. In other cases, you may not need your door to close automatically, so you wouldn't bother adding the SE 10 sprite. Figure 7.2 shows an example of a completed door in 2D view mode. The "0,20" tag in the center of the sector is

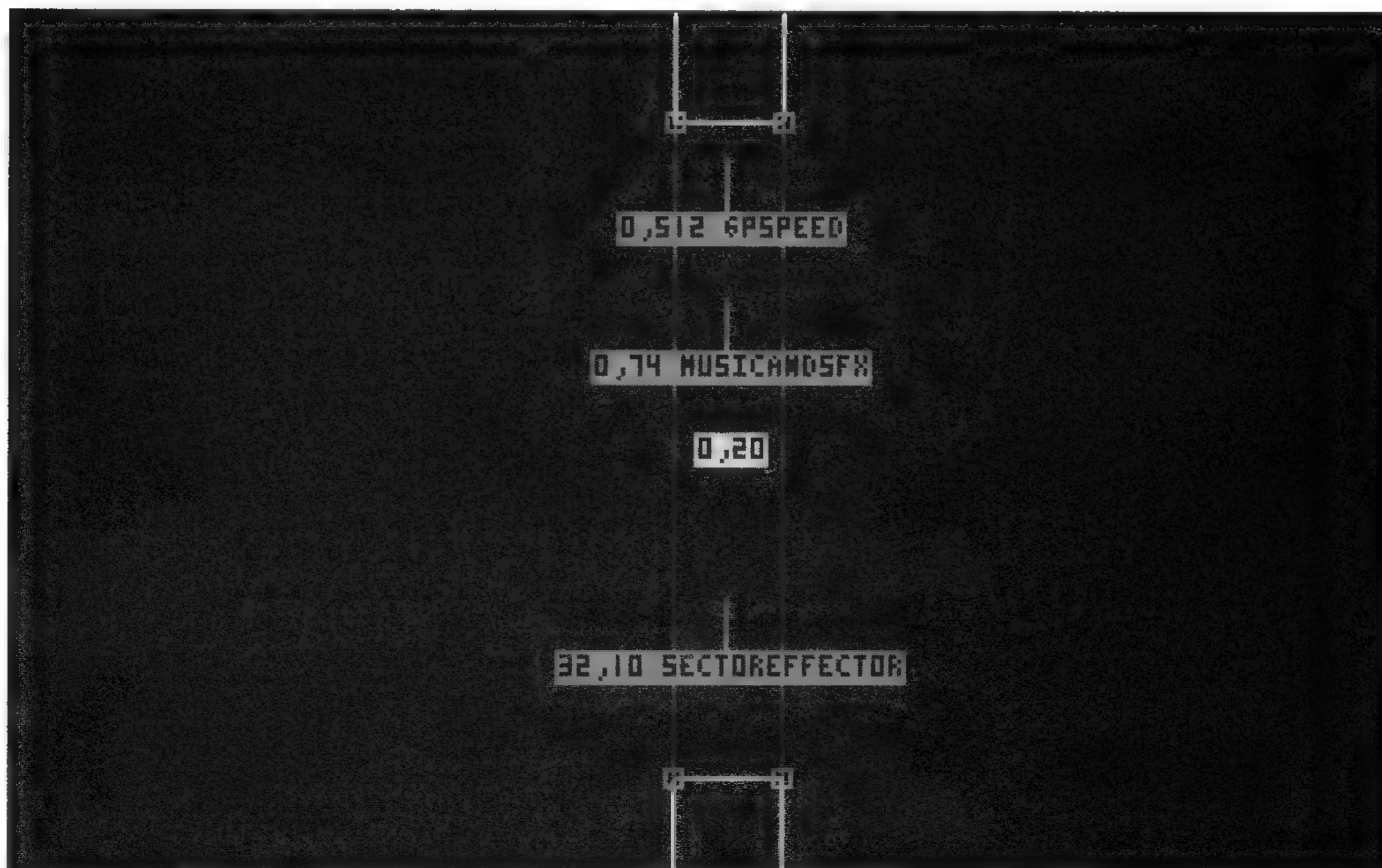


FIGURE 7.2: Here's a completed door sector in 2D view mode.

white in color and represents the sector's HiTag and LoTag values, respectively. The three other labels are the SectorEffector, Music&SFX, and GPSSpeed sprites, all with LoTags and HiTags chosen as described in the previous examples.

Figure 7.3 shows the same door sector in 3D view mode. Notice that the door sector has been raised halfway to reveal the SectorEffector, Music&SFX, and GPSSpeed sprites.

This exercise in creating a door sector provides a good introduction to the basics of creating just one type of sector special effect. Now go on to the next section, which provides a detailed list of Build's Sector Tags.

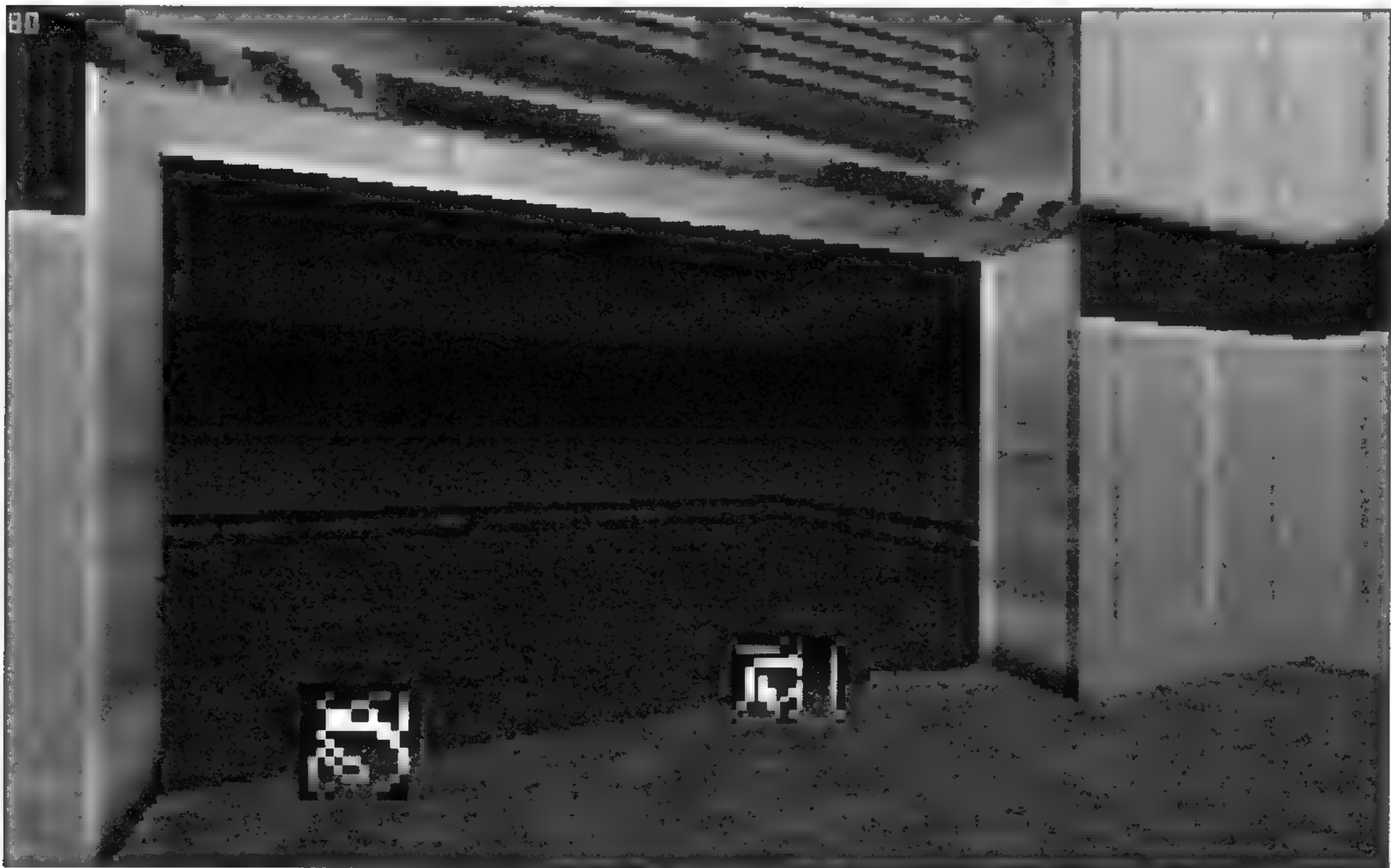


FIGURE 7.3: The completed door sector is shown here in 3D view mode.

USING SECTOR TAGS

By now you can see the general purpose of combining Sector Tags and sprites to really bring *Duke Nukem 3D* levels to life. This section provides an extensively detailed list on how to use Sector Tags along with selected types of sprites to create a variety of sector effects. In general, each appears in numerical order. Notice that Sector Tags can

be abbreviated numerically (for example, ST 1). Along with each effect is a notation of whether the effect is, in my own opinion, relatively easy, medium difficult, hard, or very difficult to create. If you desire, you can begin by reviewing the easier effects and then move on to the more difficult ones later.

THE ABOVE WATER (ST 1) AND UNDERWATER (ST 2) SECTORS

Difficulty: Medium to Hard

The ability of Duke to go underwater is actually a trick of the Build engine. The area over the water and the area under the water are actually two different sectors in two different parts of the map. The water surface acts like a teleporter, moving the player from the above water sector to the underwater sector.

To create an underwater area, start by making a two sectors of any size and shape. One of these two sectors will act as the underwater sector, and the other will act as the above water sector. Note that the two sectors must be *exactly* the same size and shape for them to behave correctly. For this reason, it might be a good idea to draw one sector first and then copy it to create the second sector. This will ensure the sectors are exactly the same.

Place one of the sectors to the left of the other. The left sector will be the above water sector, and the right sector will be the underwater sector. Refer to Figure 7.4 for a visual example.

Next, give the right (underwater) sector a LoTag value of 2. This will define the sector as ST 2, an underwater sector. To do this, place the mouse cursor in the right sector, press the T key, type the number 2, and press Enter. The sector should now have the white label “0,2” inside it. This means the sector is now marked as ST 2 (an underwater sector). If you now go into 3D mode, you will see that the sector automatically takes on a bluish palette, to suggest that the player is underwater. To enhance the effect further, you should give the ceiling of the sector texture #336, which is the water texture.

The left (above water) sector should be assigned a LoTag value of 1, which will define it as an above water sector. Because it is an above water sector, you should also make the floor of this sector look like water by giving it texture #336.

To complete the illusion of an above water and underwater area, you need to define a mechanism for objects to transport between the two sectors automatically when they break the surface of the water. This is accomplished using SE 7, which is a teleport SectorEffector. Place a new sprite in each of the two sectors. Apply texture #1 to each sprite, which makes them SectorEffector sprites. Then, give each a LoTag value of 7 to

make them SE 7 sprites. Assign a unique HiTag value that you choose (say 202 for this example). Giving each the same unique HiTag value *links* the two SE 7 sprites. An example of sectors linked in this way is shown in Figure 7.4.

Finally, you will need to move the two SectorEffector sprites so that they are in the *exact* same location in their respective sectors. The teleport action will not work if the two SE 7s are not in the exact same relative locations. To complete the area, apply texture and shade to each sector nicely, and you can also change the ceiling and floor heights of the two sectors, if you desire. Note that although the two sectors must be the same size, they can vary in altitude.

You can now attach additional underwater sectors to the left sector by giving each a LoTag value of 2. These sectors need to have a matching sector *only* if you want them to act as teleporters to a corresponding above water sector. If you would like a below water tunnel attached to the first underwater sector, similar to the one shown in Figure 7.5, it would not need to have a matching above water sector. Note that you would probably not give the tunnel a ceiling with a water texture, however, because the player would think he could surface in this area. Just make sure your linking sectors remain the exact same size as you alter the underwater texture to add such tunnels. That is, if you add vertices to a wall to create a new attached sector, make sure to add the same vertices to the above water sector so that the two remain the same size.



TIP

Now that you've used number 202 to link two sprites, you shouldn't use this number again anywhere else on your map. It's a good idea to keep a written log of each unique value that you choose, and where you used each one, so you won't accidentally reuse a value later on.

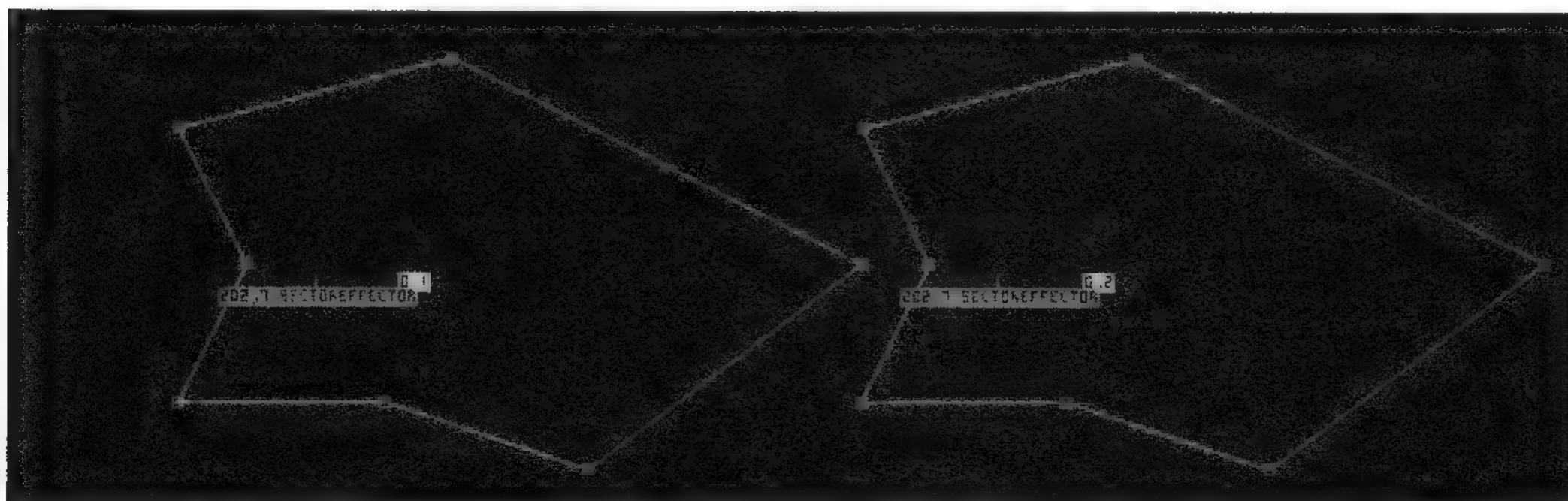


FIGURE 7.4: These two sectors are linked to act as an above water and an underwater area.

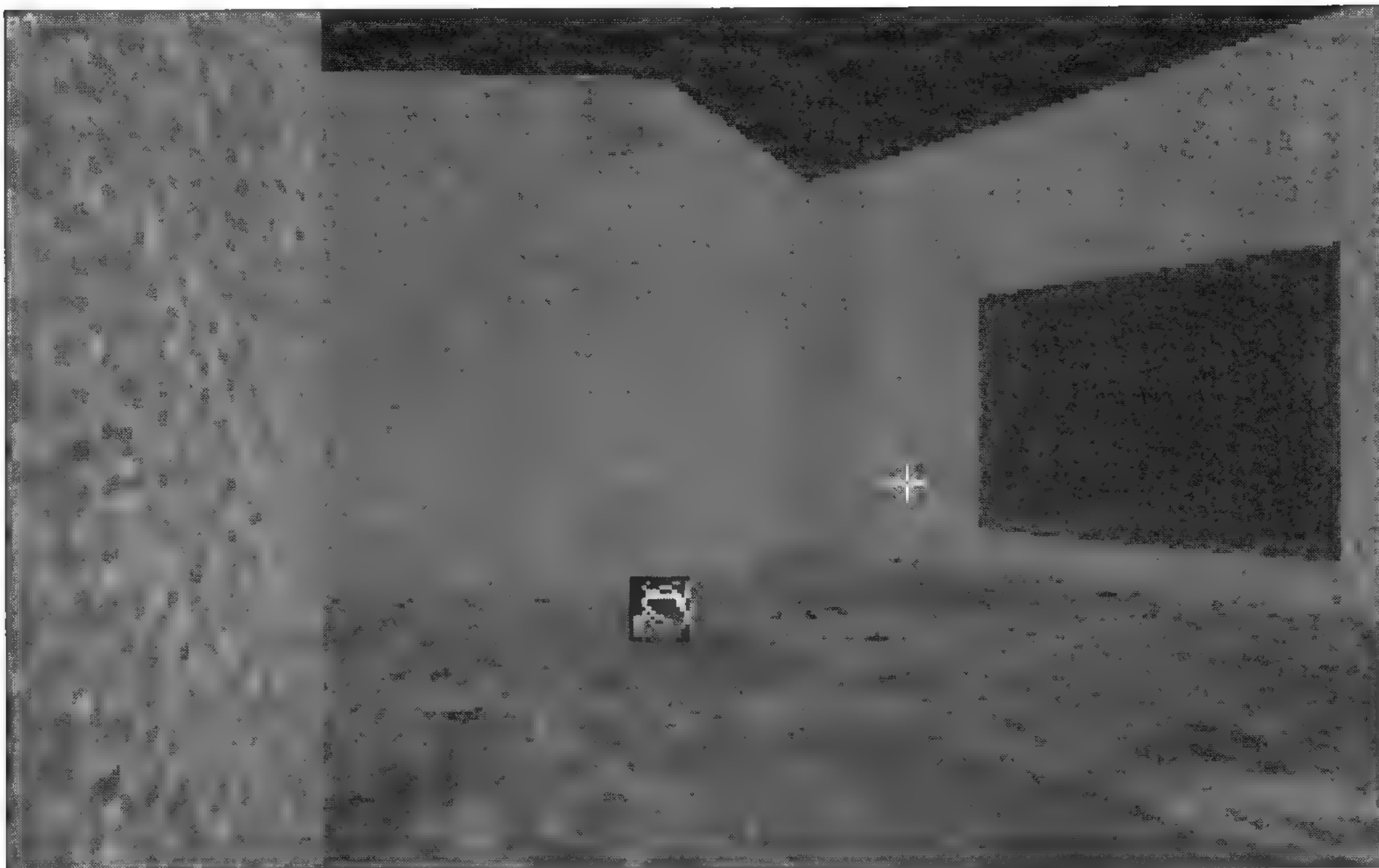


FIGURE 7.5: In this 3D view from underwater, notice the underwater tunnel sector on the right-hand side of the screen, with a ceiling that has a solid texture as opposed to a water texture.

Using an Underwater Sector (ST 2) Alone

Difficulty: Medium to Hard

Keep in mind that you need to create a ST 1 and ST 2 pair *only* if you want to provide a means for the player to move from underwater to above water. Any sector can be given a LoTag value of 2, and it will act like an underwater sector. You can then make all sorts of underwater tunnels and caves and secret rooms without creating corresponding above water sectors. Just make sure each one of these sectors is somehow connected to a sector that will allow the player back to the surface, or Duke will run out of air in short order!

STAR TREK DOORS (ST 9)

Difficulty: Easy to Medium

The Star Trek door is a door that splits vertically and opens by spreading outward, like the doors in the original “Star Trek” TV show. It also differs from other doors in

that the wall textures on the door compress when the door is opened, giving the appearance that the door isn't solid. These types of doors are also used for opening curtains, like the curtains in the movie theater in *Hollywood Holocaust* (E1L1). Here are the basic steps for making a Star Trek door. They are harder to make than the regular *DOOM*-style door you did previously.

Start with two room sectors and a smaller sector in between, as you did with the *DOOM*-style door. Split each of the short walls of the door sector by adding a vertex. Drag these new vertices toward each other a few grid lines, as shown in Figure 7.6.

Split the four diagonal lines that extend from the two vertices you just dragged together. Place these four new vertices so they're on the same line as the longer two walls of the door sector, as shown in Figure 7.7.

Then, drag the two center vertices a bit more until they touch in the exact center of the door sector. The door sector now appears as two triangles with their points touching, as shown in Figure 7.8.

Place the mouse cursor within one of these two triangles and press the T key to give the door sector a LoTag value. Type the value 9, which represents the Star Trek doors. Add a Music&SFX sprite and a GPSpeed sprite, as you did with the *DOOM*-style door, if you desire. Note that for some reason, the sound effect you choose with the Music&SFX

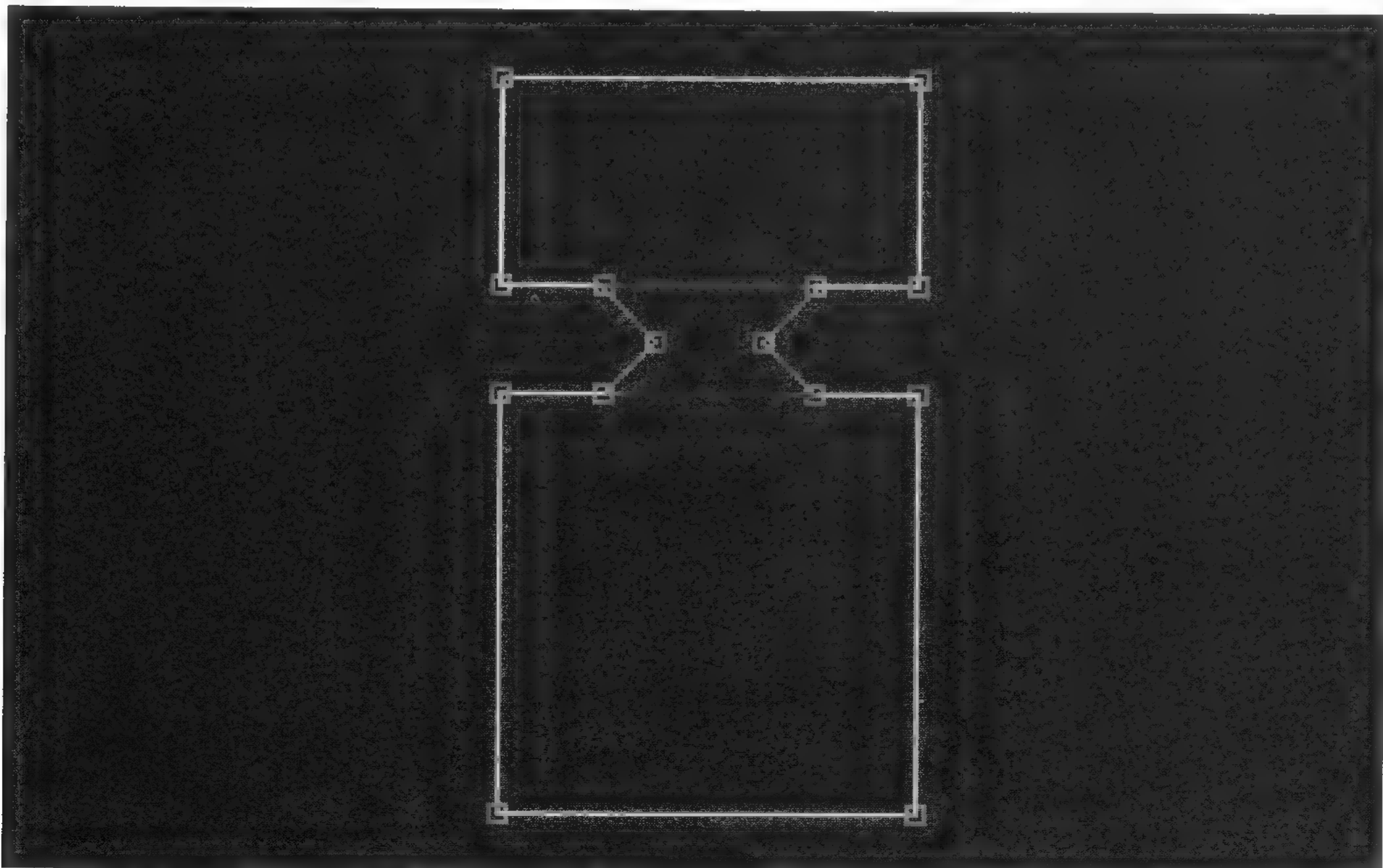


FIGURE 7.6: The Star Trek door's sector is started by inserting two vertices and dragging them toward each other.

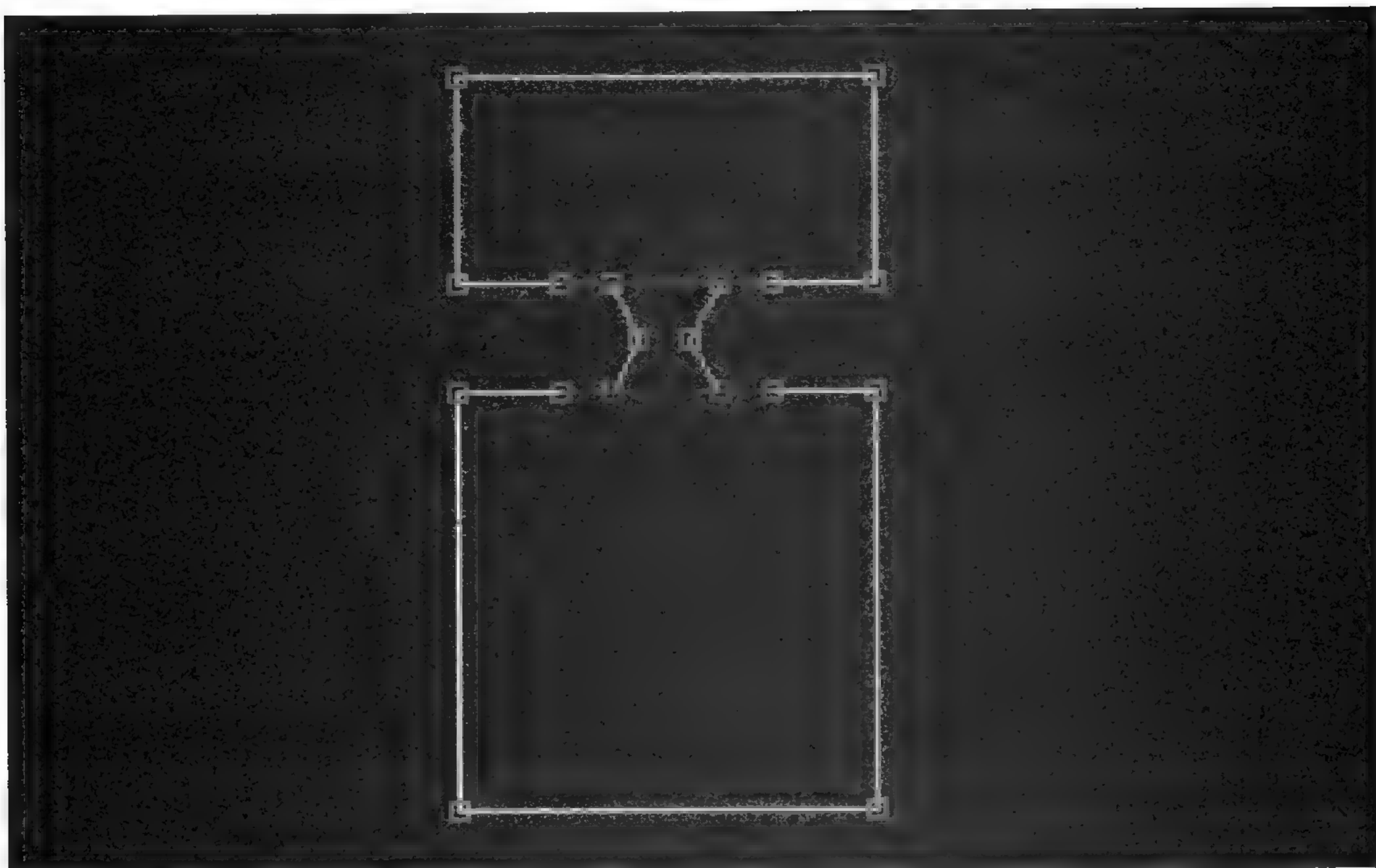


FIGURE 7.7: Move the four new vertices into place like this.

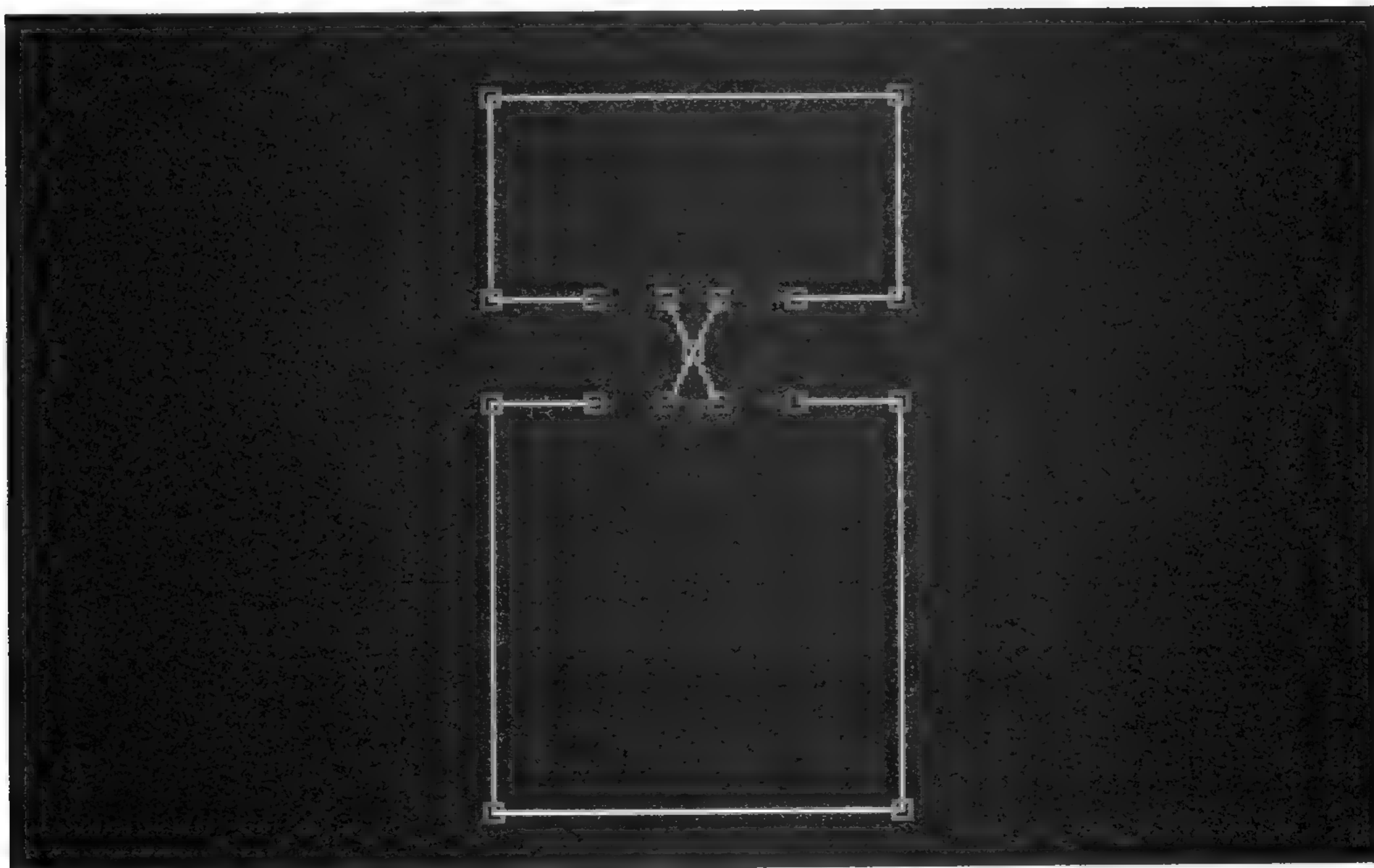


FIGURE 7.8: The correct orientation for the vertices for the Star Trek doors is shown here.

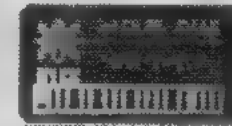
sprite will play twice for this type of door. This somewhat undesirable effect seems to be built into the game and cannot be avoided. The only choice is to live with the sound playing twice or make the door silent by not adding the Music&SFX sprite.

ELEVATOR TRANSPORT (ST 15)

Difficulty: Very Difficult

This type of elevator is a special effect that actually transports the player between two different elevator shafts. The illusion is that the elevator is in one long shaft, however. This allows you to create complex areas that appear to be two story areas, but can actually be in two totally separate areas of the map.

Start by creating three sectors as shown in Figure 7.9. The western sector will be the elevator car; the eastern sector is the room in



NOTE

I experienced a great deal of difficulty in getting the elevator transport effect to work, which is why I label it a *very difficult* effect to create. However, if you follow the directions in this section, you should be able to get one working in no time.

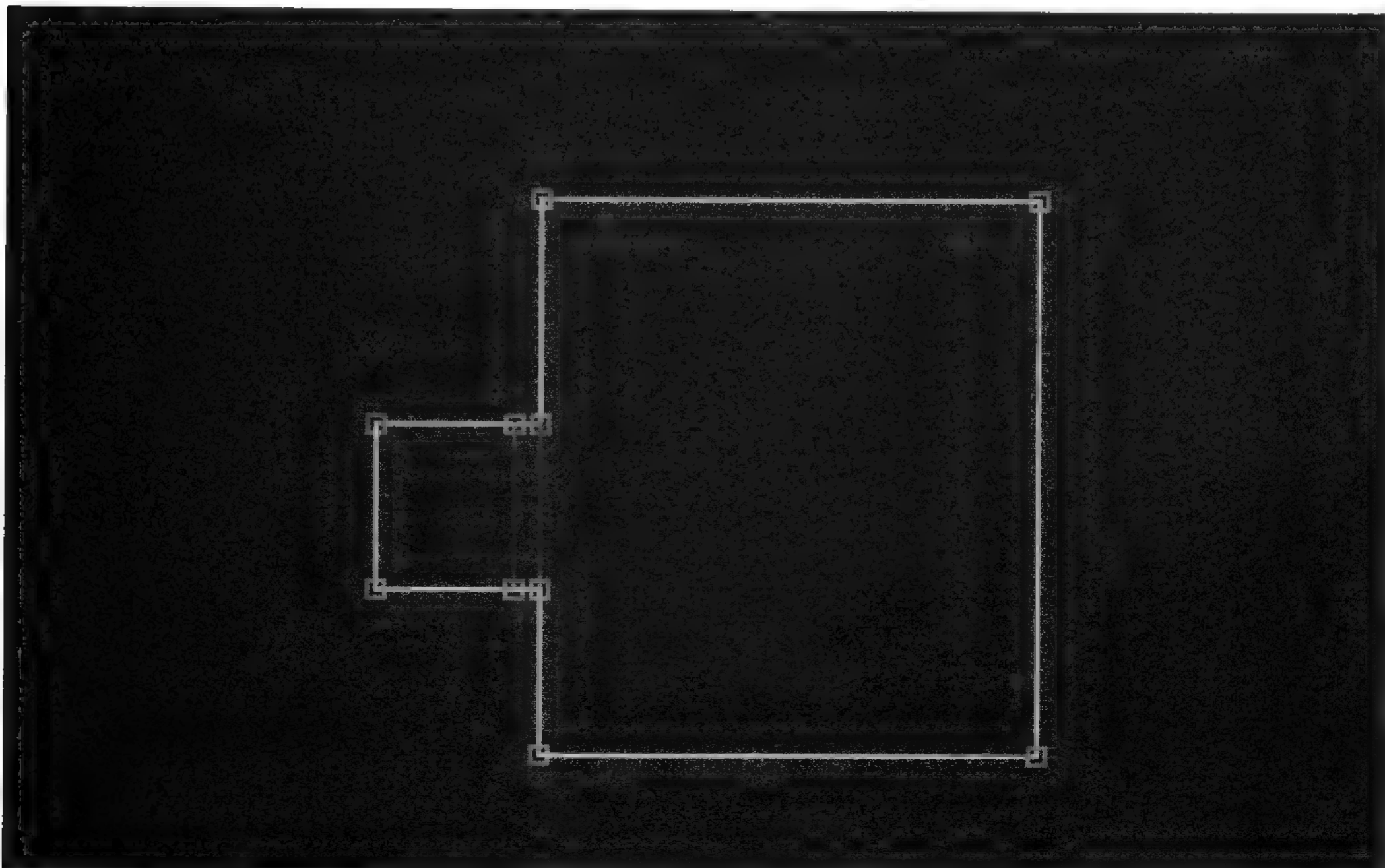


FIGURE 7.9: Start the elevator transport effect by creating three sectors: one each for the elevator car, the threshold, and room the car starts in.

which the car will start. The middle sector acts as a threshold between the room and the car.

Next, switch to 3D view mode and lower the ceiling of both the elevator sector and the threshold sector a few clicks, so that the car sector is shorter than the door sector,

as shown in Figure 7.10. This step is *extremely* important, as an elevator car that starts off the same height as its parent sector won't function correctly for some reason.

Now, you need to create a second area where the elevator transport will bring the player. This is best done by selecting all three sectors and copying them to a new area of the

map. For this exercise, move the new sectors just to the left of the original ones, as shown in Figure 7.11.

For the elevator transport to work properly, the two elevator cars *must* be at the proper physical altitudes. That is, one car has to start off much higher than the other car. In addition, there needs to be a full car height of space *in between* the ceiling of the lower car and the floor of the higher car. See Figure 7.12 for an illustration of this.

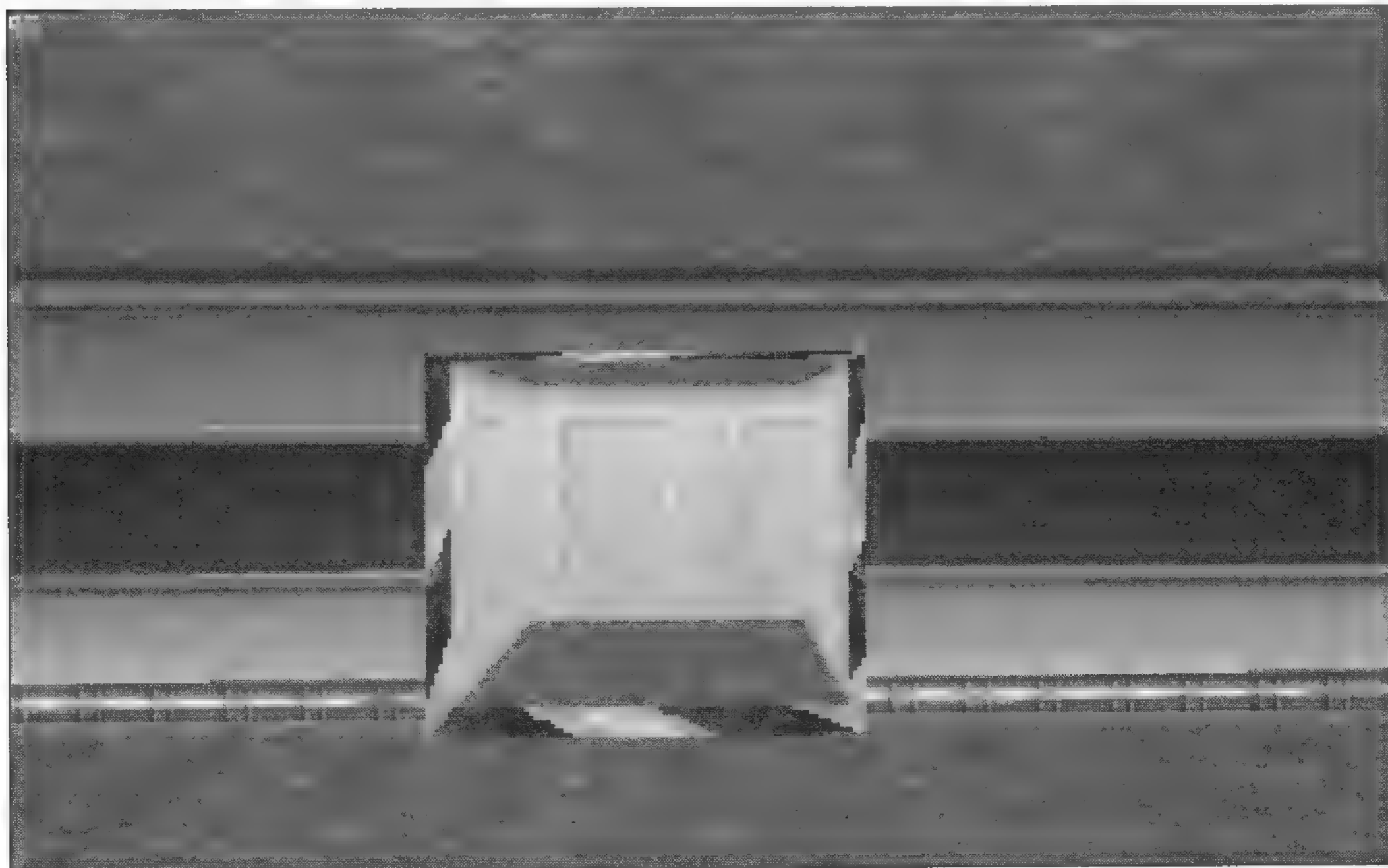
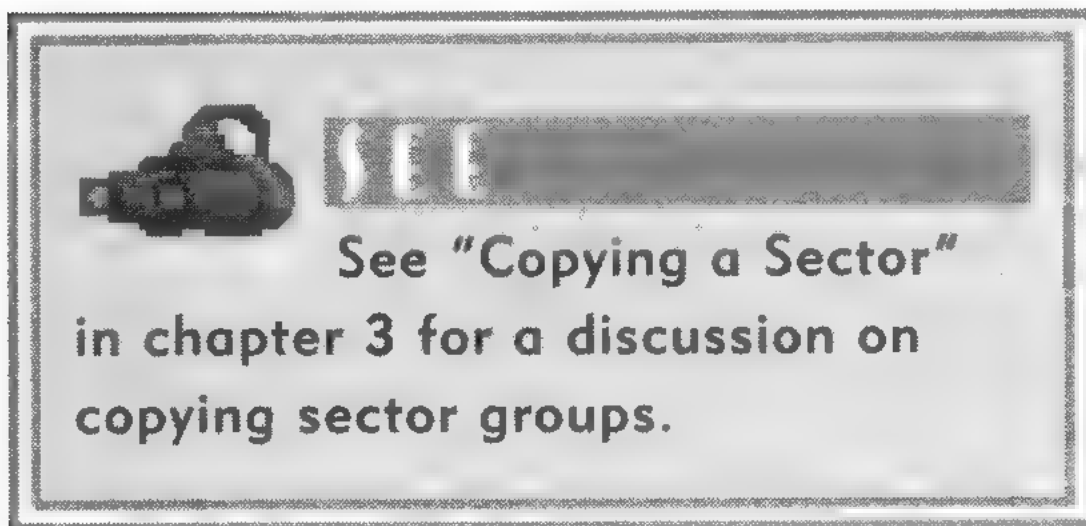


FIGURE 7.10: Lower the ceilings of the elevator car and threshold sectors; this step is vital to the proper functioning of the elevator transport effect.

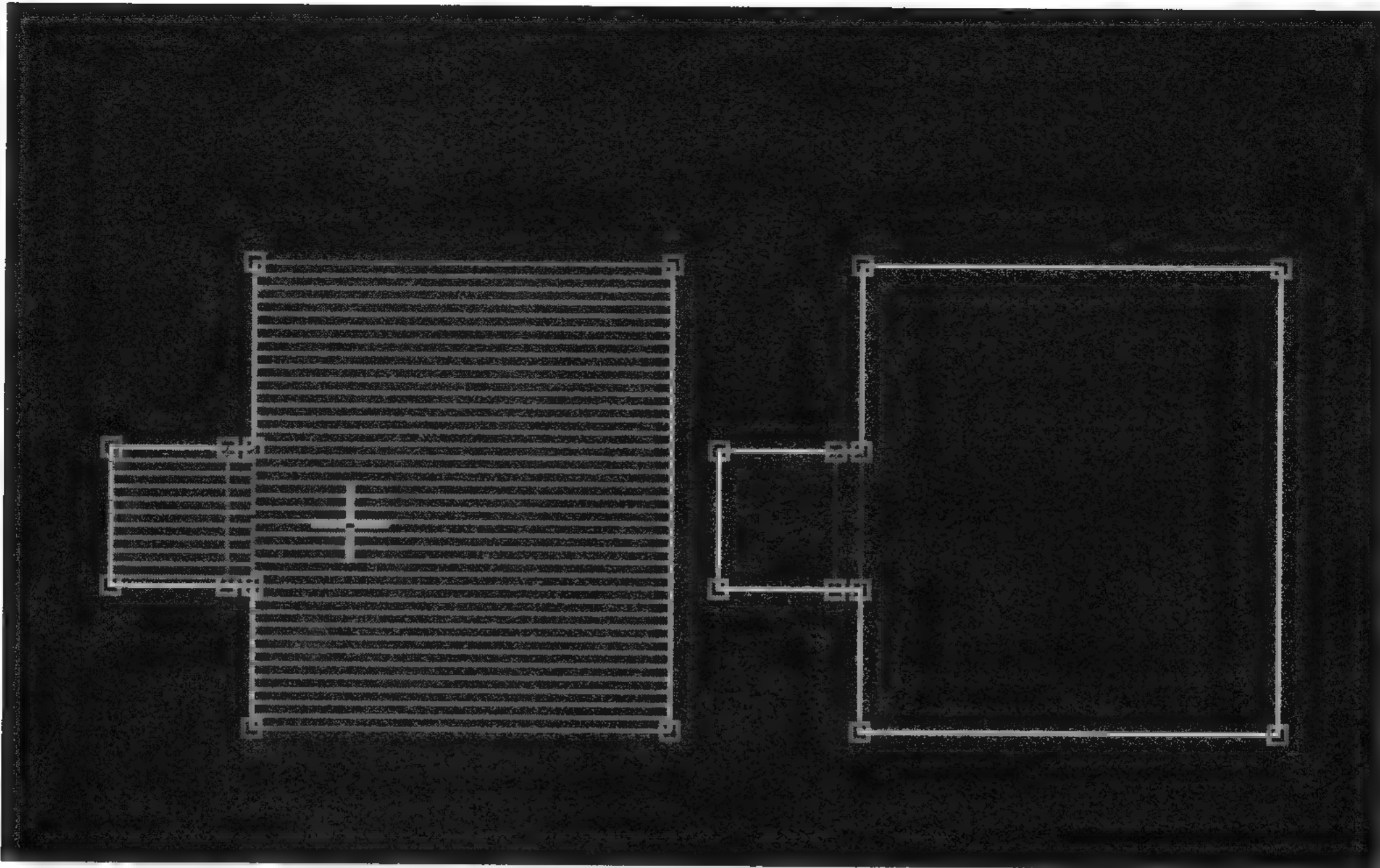


FIGURE 7.11: Copy the three sectors to a new area to the left.

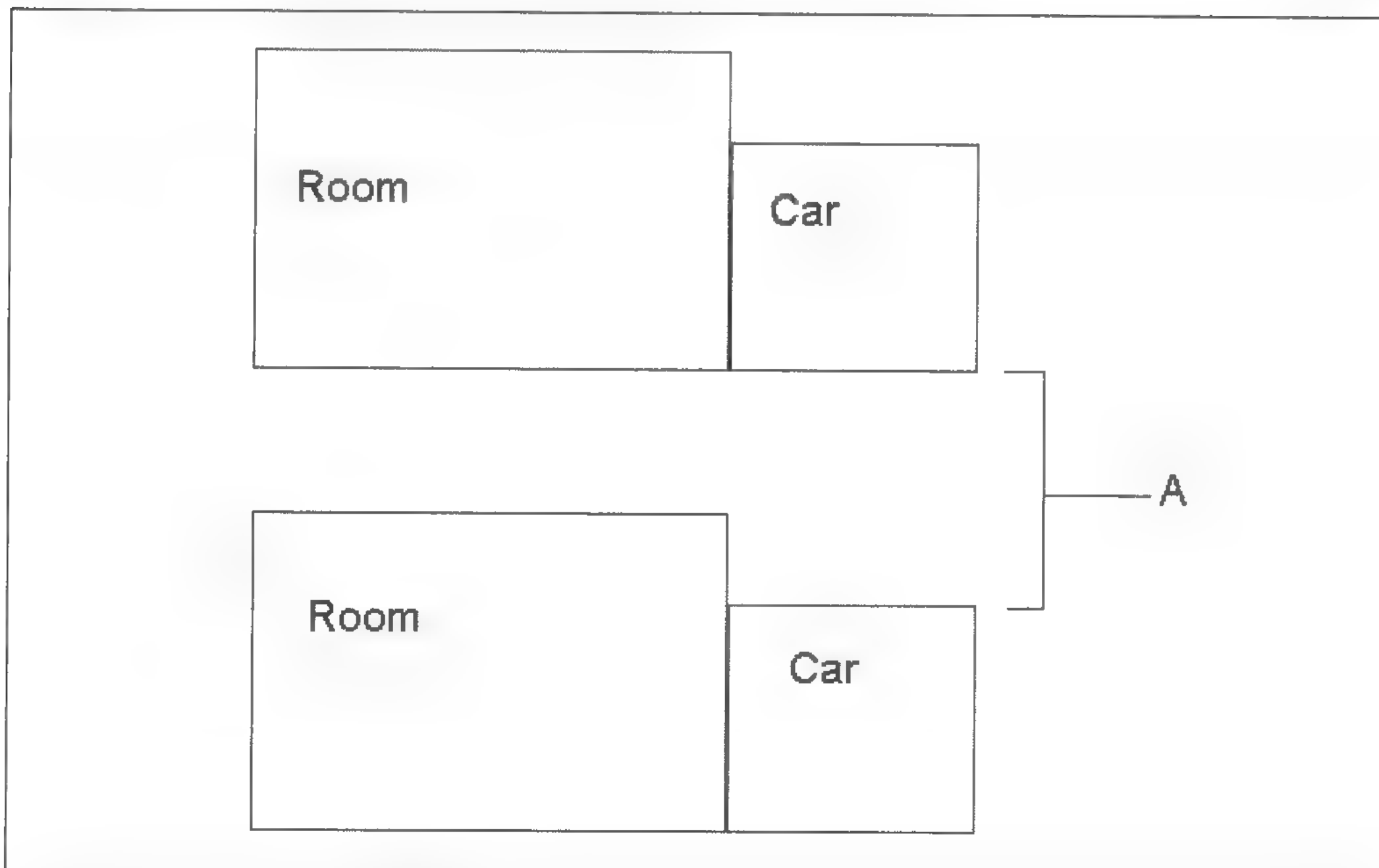


FIGURE 7.12: The two elevator cars need to have a space of at least one car height between them in order to work properly.

Because there is not a good tool in Build to easily see how high each car is relative to each other, it's better to be safe than sorry when changing the height of the car sectors. I chose to use a height difference of 30 clicks (30 presses of the PgUp or PgDn key) between the two elevator cars. To make one car lower than the other, place the mouse cursor on the floor of the car, hold down the left mouse button, and press PgDn 10 times. Then select the ceiling of the same sector, and lower it 10 clicks as well. Repeat this process. Make sure to lower the ceilings and floors of all three sectors: the car, the threshold, and the room sectors. When you finish, your room should look exactly like it used to, except that it's physically lower on the z axis (which you can't see right now).

The next step is to give each elevator sector a LoTag value of 15, which is the value for the Elevator Transport Sector Tag. Also, give the higher of the two elevator car sectors a HiTag value of 1 (leave the other elevator's HiTag value at 0).

Now, to create the transport effect, you need to place a SectorEffector 17 in each of the elevator cars. The SectorEffectors must be in the same place in each car, or Duke will die when he transports from one elevator to another. Give each of these two SectorEffectors a LoTag value of 17, and a unique, unused HiTag value (say 203 for this example). Finally, make either one of these two SectorEffector sprites a darker shade (consider using shade 32, which is totally black). The sprite that has the darker shade represents the floor that the elevator car will start on.

Figure 7.13 shows the completed elevator transport car. You can now go and try it out in the game.

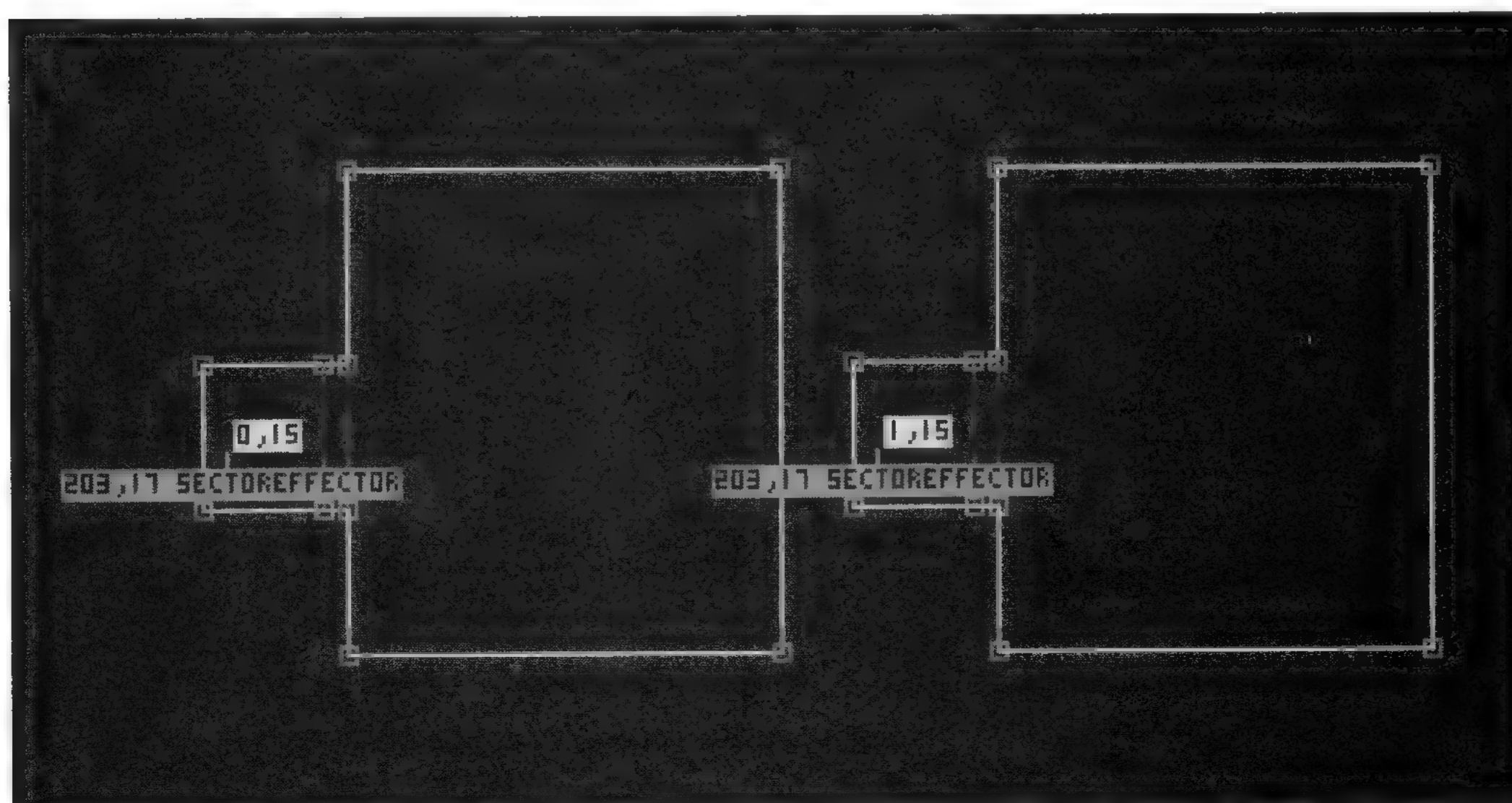


FIGURE 7.13: Here is the finished elevator transport effect in 2D view mode.

ELEVATOR PLATFORM DOWN (ST 16)

Difficulty: Easy

In contrast to the elevator transport just discussed, *standard* elevators are very easy to make in Build. There are four types of these more conventional elevators, and they all accomplish basically the same thing: lowering and raising a player between sectors with different floor heights. Follow these directions to create an elevator platform going down.

Start by making two simple room sectors separated by a few grid lines. Add a small square sector in between the two, as shown in Figure 7.14. The middle sector will become the elevator platform.

Put the player (white arrow) in the northern room, and switch to 3D view mode. Raise the ceiling of this sector by placing the mouse cursor on the ceiling, holding down the left mouse button, and pressing PgUp 15 times. Then, place the mouse cursor on the floor of the same room, hold down the left mouse button, and press PgUp another 15 times. This raises the room so that it is much higher than the other room, and it gives your elevator room to go up and down.

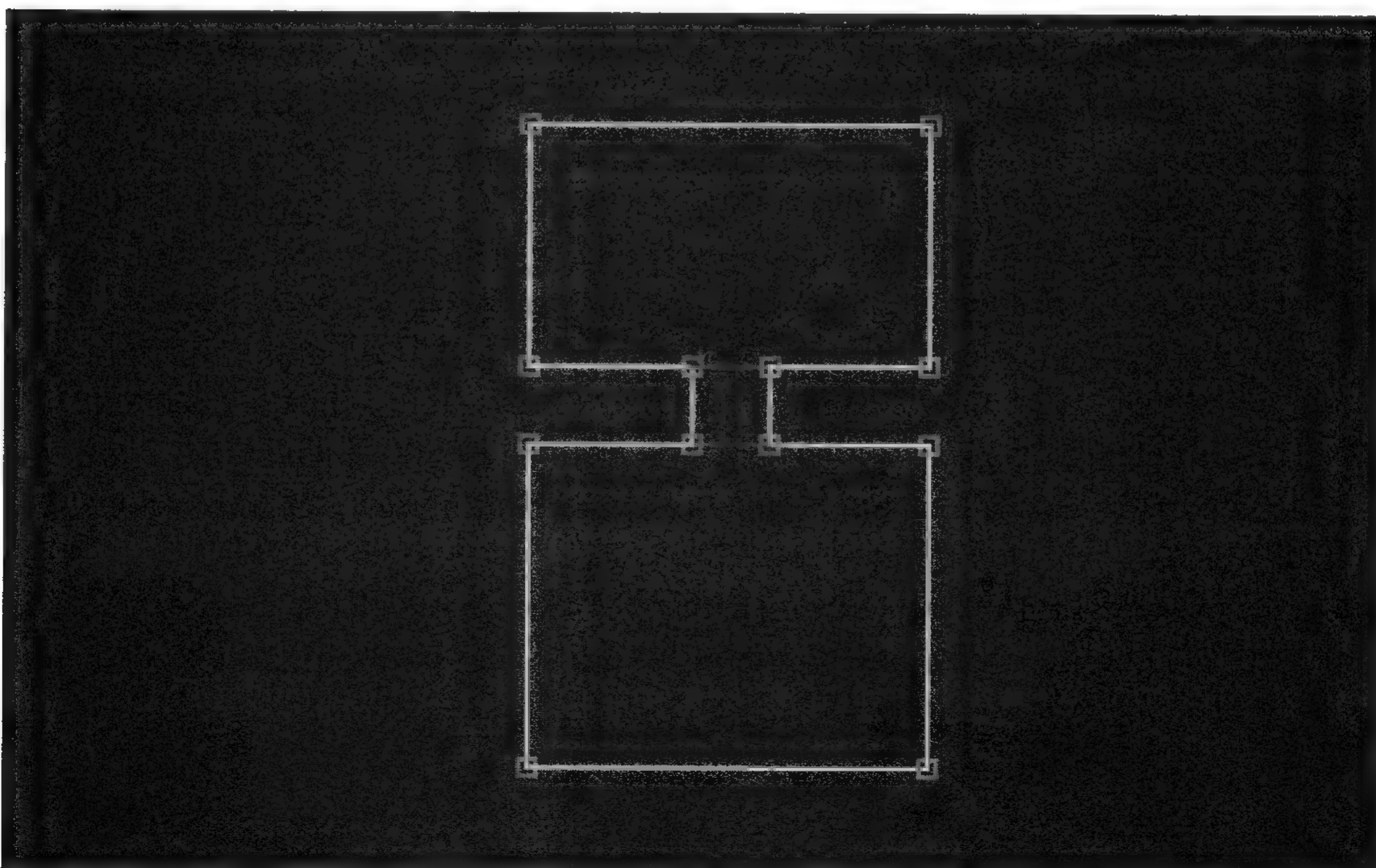
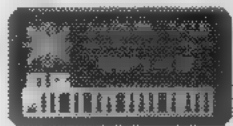


FIGURE 7.14: Create a middle sector for an elevator platform between two room sectors.

**NOTE**

The number 15 was chosen somewhat arbitrarily for the purpose of this example. Your elevators can span any distance between floors that you choose.

Now, point the mouse cursor at the ceiling of the elevator sector and raise its ceiling and floor up 15 clicks as well. This makes the elevator sector's floor and ceiling the same height as the higher of the two rooms. Give the elevator sector a LoTag value of 16, which is the sector tag number for the elevator platform down effect. This elevator starts in the up position and moves downward until it encounters

a neighbor sector with a lower floor than where it started. That's all there is to it! Your lift should work fine in the game now.

You can also spruce up this effect by adding a Music&SFX sprite to the elevator sector to control the sound the elevator makes when it moves. You can do this by assigning a LoTag value for the Music&SFX sprite equal to the number of the sound you want the elevator to make when it starts, and assign the HiTag value of the same sprite to the number for the sound the elevator will make when it stops. Finally, you can add a GPSSpeed sprite to control the elevator's rate of movement. The LoTag value for the GPSSpeed sprite will control the speed of the elevator.

ELEVATOR PLATFORM UP (ST 17)

Difficulty: Easy

This effect is exactly the same as the elevator platform down effect, except that the lift should start in the lower of the two sectors instead of the higher of the two.

To make this effect, create the same sector arrangement as before (see Figure 7.14). Raise the ceiling and floor of one of the rooms several clicks so that it's higher than the other room. As for the elevator sector, raise the ceiling of the sector to match the ceiling of the higher room. *Do not* adjust the floor of the elevator sector; you want the floor to be even with the floor of the lower room. Then, give the elevator sector a LoTag value of 17, which is equivalent to the Elevator Platform Up Sector Tag number.

As with ST 16, you can give the elevator sector a Music&SFX sprite to make the elevator make sounds as it starts and stops and a GPSSpeed sprite to control the rate of movement of the elevator.

ELEVATOR CAR DOWN (ST 18)

Difficulty: Easy

This third type of elevator works exactly like the ST 16 effect, except that the floor *and* the ceiling of the elevator move, making the elevator behave more like a real elevator car rather than just a moving platform.

To create this sector effect, follow the exact steps described for the ST 16 effect, except give the elevator sector a LoTag value of 18 instead of 16. The elevator will perform the same way, except that the ceiling of the elevator car will move as well as the floor. As with all the elevator effects, you can add a Music&SFX sprite and a GPSSpeed sprite to further define the elevator's behavior.

ELEVATOR CAR UP (ST 19)

Difficulty: Easy

The last type of elevator, as you might guess, is an elevator car that starts in the down position and moves upward. The floor and the ceiling of the elevator sector move. To create this type of elevator, make your three sectors exactly as before. Your elevator sector should have the same ceiling and floor as the lower of the two rooms. (You might want to lower the ceiling of the elevator car just a bit to make it lower than the room it starts off in.) Then, give the elevator sector a LoTag value of 19. Add a Music&SFX sprite and a GPSSpeed sprite, if you desire.

CEILING DOOR (ST 20)

Difficulty: Easy

You already experienced using ST 20 extensively, if you worked through the exercise to create the first door (*DOOM*-style) in this chapter.

FLOOR DOOR (ST 21)

Difficulty: Easy

If you have successfully made ST 20, the ceiling-based door effect, then this floor-based door effect will be very easy to create. Either go to the area of the map where you made the ceiling-based door, or if you've already deleted it, make the ceiling-based door again. There is an interesting trick to be learned in converting a ceiling-based door to



SEE

See "Creating a Simple *DOOM*-style Door" on page 137 for information on using ST20 to create this effect.

a floor-based one, so take a look at how to do it this way rather than simply creating the floor-based door from scratch. You can change your existing ceiling-based door to a floor-based one in a few short steps.

Switch to 2D view mode, put the mouse cursor within the door sector, and press the T key to change the sector's LoTag value from 20 to 21, which is the Sector Tag number for a floor-based door. Now all you have to do is move the door's ceiling and floor so that they both start at the *top* of the sector instead of at the bottom, and your door will be done. You need to switch to 3D view mode to do this.

At this point, you may notice an interesting dilemma. The way that you move a sector's ceiling and floor height is by placing the mouse cursor on the ceiling or floor and pressing the PgUp or PgDn key. However, right now your door sector should be completely hiding its ceiling and floor because they meet! How on earth can you place the mouse cursor on the ceiling or floor when you can't see it? The answer is to use an interesting maneuver, one that you'll have to learn if you ever close a door in Build and then need to reopen it. Use the following steps to open the door so that you will be able to see its sector's ceiling and floor:

1. Switch to 3D view mode, and put yourself directly in front of the closed door.
2. Turn 90 degrees to the left or right.
3. Press the numeric keypad's Ins and Del keys. These keys will slide you to

the left and right. Keep sliding until you're *inside* the door sector. Your screen should look like a bunch of junk at this point because Build doesn't know how to draw your view when you're standing inside a sector that effectively has zero height. In some cases, if you land in just the right spot, the screen will be two solid colors divided horizontally across the center.



NOTE

Make sure to use the numeric keypad's Ins and Del keys. Their counterpart keys in the other set (near the Home and End keys) do not work for this operation.

4. Place the mouse cursor near the top center of the screen. Hold down the left mouse button, and press PgUp several times. This should open the door a bit. Your view will change to a more conventional one if you accomplish this successfully. You can now move outside the door sector and open it up the rest of the way.

This trick takes a bit of practice, but it's necessary for opening up sectors that have been completely closed by joining their ceilings and floors. Try it a few times until you get used to it.

Once you're satisfied you can get any closed door open again, bring both the ceiling and the floor of the door's sector up to match the ceiling of one of the adjoining rooms. That's it! Go into the game and try out the floor-based door. You'll probably notice that the side walls are once again moving when the door moves. You'll have to realign these again using the O key to align them to the top of the sector.

Creating a Floor-Based Door from Scratch

Of course, you can create a floor-based door from scratch. Figure 7.15 shows a floor-based door. Notice the striped threshold sector in front of the door. This type of sector is often used to lower the ceiling near the door when the door would normally face a very tall sector, and it allows the door to remain a more realistic size.

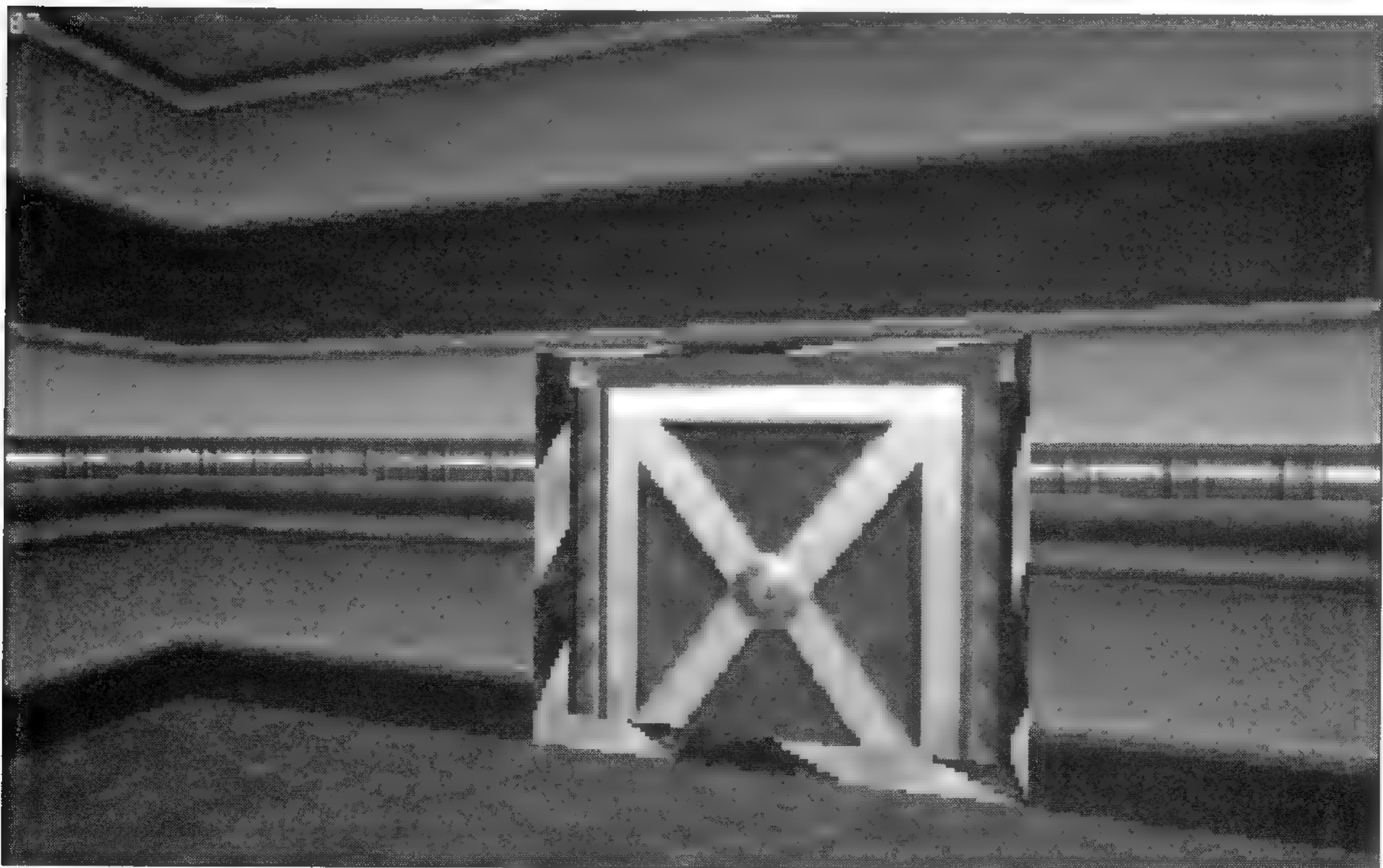


FIGURE 7.15: This floor-based door is open just a bit to show that it opens from the top.

To make a floor-based door sector *without* copying the ceiling-based door first, follow these steps:

1. Make a thin sector bordering two existing sectors.
2. Make the sector's LoTag value 21.
3. Add a Music&SFX sprite (5) for the sound.
4. Bring the door's floor up to the ceiling.
5. Add a GPSPeed sprite (10) to change the speed.
6. Add a SectorEffector type 10 sprite to make the door close automatically.

THE SPLIT DOOR (ST 22)

Difficulty: Easy

Creating a split door effect is just like creating the ceiling- and floor-based doors, except that the split door opens from both the ceiling and the floor, with a *crack* in the middle of the door. To change your floor-based door to a door of this type, switch to 2D view mode, and change the sector LoTag value to 22. Then move the ceiling and floor to the center, halfway between the ceiling and floor of one of the adjoining room sectors. The final product should look like the one shown in Figure 7.16.

THE SWINGING DOOR (ST 23)

Difficulty: Medium to Hard

These types of doors are much different than the doors discussed previously, so you'll have to start over in a new area of the map. Swinging doors can come in pairs, like the entrance to the theater in Hollywood Holocaust (E1L1), or they can be single, like a bathroom stall entrance or a locker door. You'll practice making a pair of swinging doors here.

Start off with a square room. On the northern wall, make another sector off of the first room. This will be a hallway into the room. The pair of swinging doors will join the room to the hallway.

The first thing you need to decide is which direction the swinging doors will swing: either into the room or into the hallway. Swinging doors are different from the other doors you've worked with because their door sector must lie *inside* another sector,

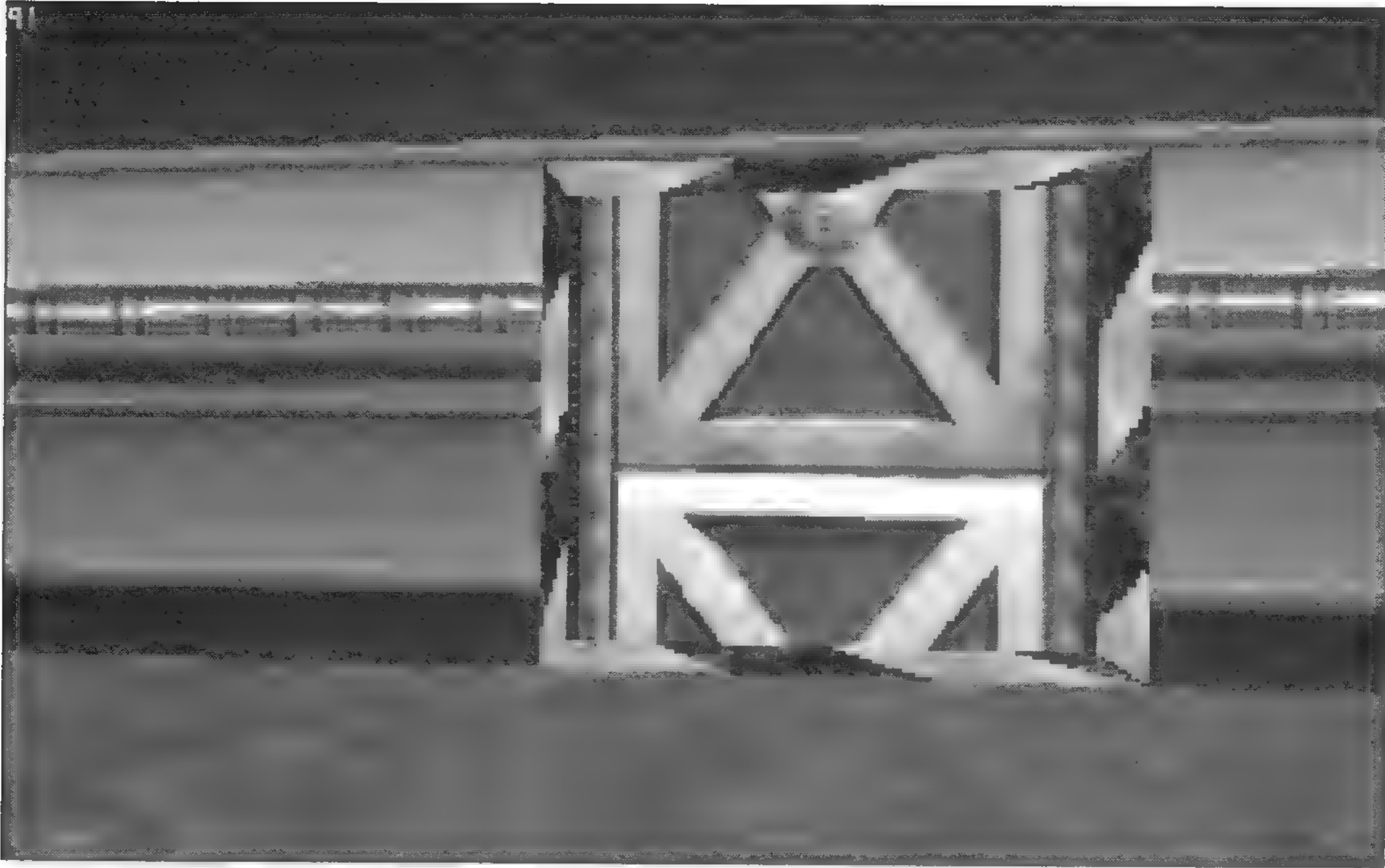


FIGURE 7.16: The split door effect provides a door that opens from the floor and ceiling.

whereas the doors discussed previously were made from stand-alone sectors. The sector the door will swing into is the sector in which you must put the swinging door's sector. If a swinging door sector is a child of one sector but swings into another, the Build engine will not draw the door correctly.

For this example, make the door swing into the room by making the swinging door's sector a child sector within the parent room sector. Accomplish this by drawing a thin rectangular sector inside the room sector, with the long way pointing from west to east. Then, place the mouse cursor inside this child sector and press Alt + S to make the walls of the new sector two-sided. Split the north and south walls of the child sector with new vertices (Ins key). Finally, move the door sectors so that the northern vertices are right on top of the line that joins the room sector to the hall sector, and the southern vertices still lie inside the room sector. Your map should look something like Figure 7.17.

The swinging door is ST 23. In 2D view mode, place the mouse cursor inside each of the two door sectors, press the T key, type 23, and press Enter.

Now you need to define a pivot point around which each door will rotate. The pivot point of a swinging door is SectorEffector 11. Place a SectorEffector sprite (1) near the northwest vertex of the west door and near the northeast vertex of the east door. (Make

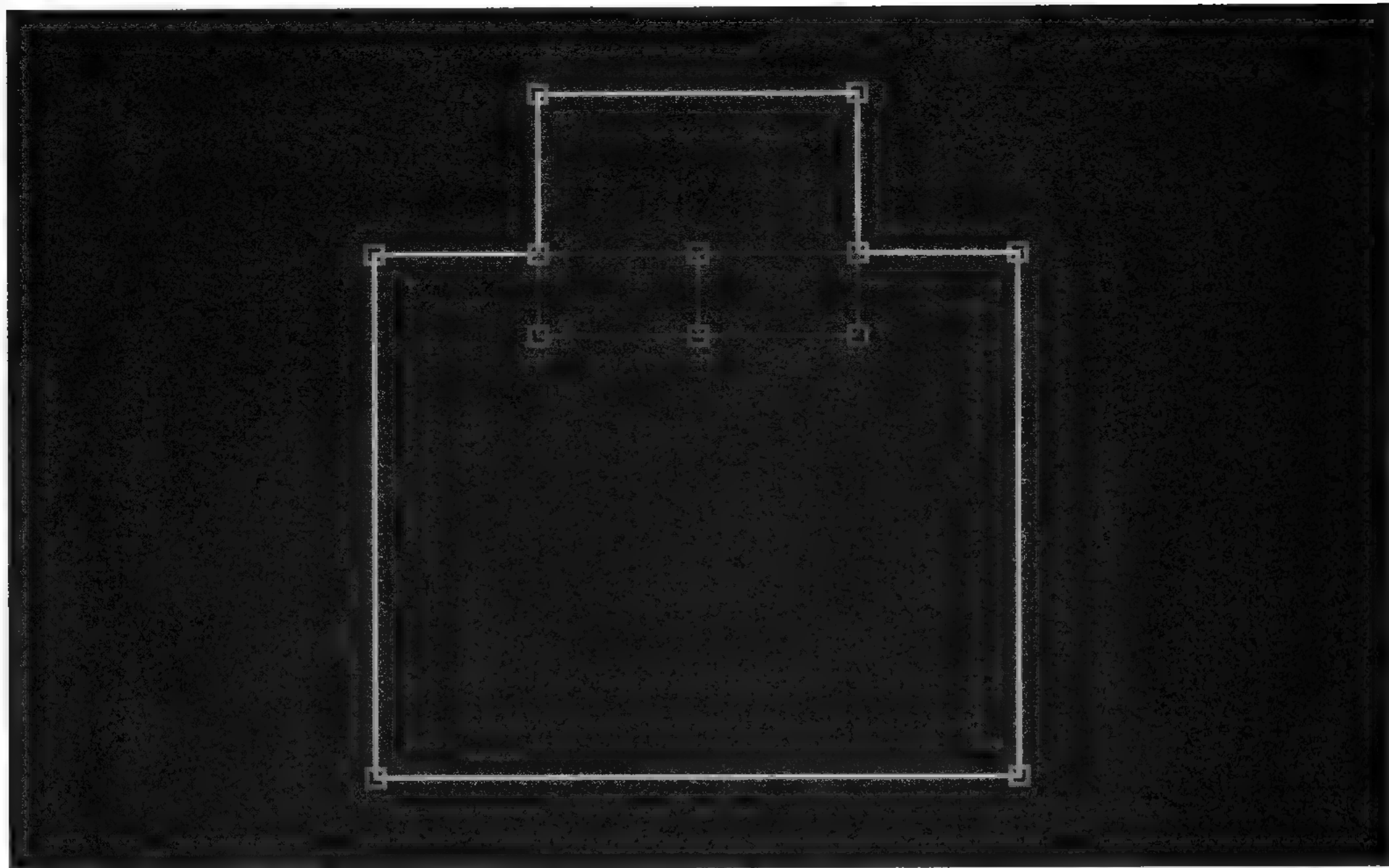


FIGURE 7.17: For swinging doors, create a child sector within the sector in which you want the swinging doors to swing.

sure to keep the sprites inside, not on, the borders of the door sectors. Turn the grid off temporarily to do this.) For each SectorEffector sprite, place the mouse cursor on it, press Alt + T, type 11, and then press Enter. This gives each a LoTag value of 11.

You can also give these sprites a HiTag value. Giving them the same HiTag value will make the doors swing open and close together; giving them a different HiTag value (or leaving the HiTag value at 0) will cause the doors to open and close independently. The angle of this sprite controls the swing direction of the door—up (north) for clockwise or down (south) for counterclockwise. You want the western door to swing clockwise and the eastern door to swing counterclockwise. Use the , (comma) and . (period) keys to control the angle of the sprite. Make the angle point up for the western SectorEffector and down for the eastern SectorEffector.

As you did with the other types of doors, you can add a Music&SFX sprite to give the swinging doors a sound and a GPSSpeed sprite to alter the speed of the doors. To complete the swinging doors sectors, switch to 3D view mode and bring the sectors' ceilings down to the level of the floor. The final product should look similar to the one shown in Figure 7.18. You should now have a pair of swinging doors! Go into the game and try them out.

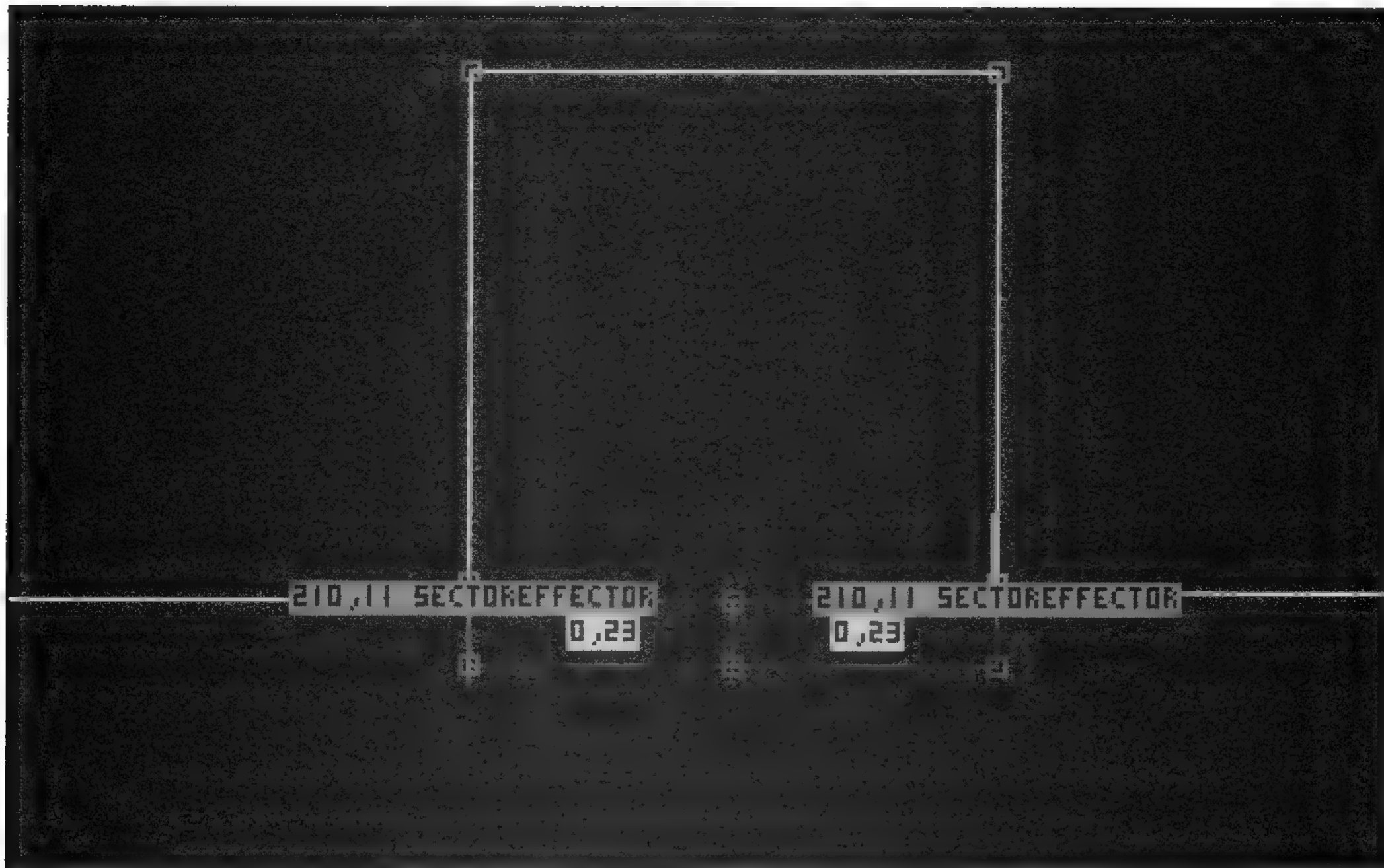


FIGURE 7.18: Here are the final swinging doors sectors.

THE SLIDE DOOR (ST 25)

Difficulty: Medium

A slide door looks like a Star Trek door, except that the door textures don't squish. They are a bit more complex to make than the Star Trek doors, however.

Start with the two rooms and the thin door sector, as with most of the other door effects. Split the two shorter walls of the door sector by adding three new vertices, and start to pull the center vertices inward, just as you would do for a Star Trek door. However, don't move them to the exact center just yet. You want to split the door sector into two sectors, as shown in Figure 7.19.

After you've split the door sector into two, move the three new vertices in each sector into place as you did for the Star Trek door. Then, give each of the door sectors a LoTag value of 25.

The last step for this door effect is to define two SectorEffector 15 sprites, and place them in the two door sectors. These sprites control the direction of the sliding door. Assign each sector a LoTag value of 15. Change the angle of the sprite to point in the direction the door will move when it *closes*. That is, the western door's sprite should

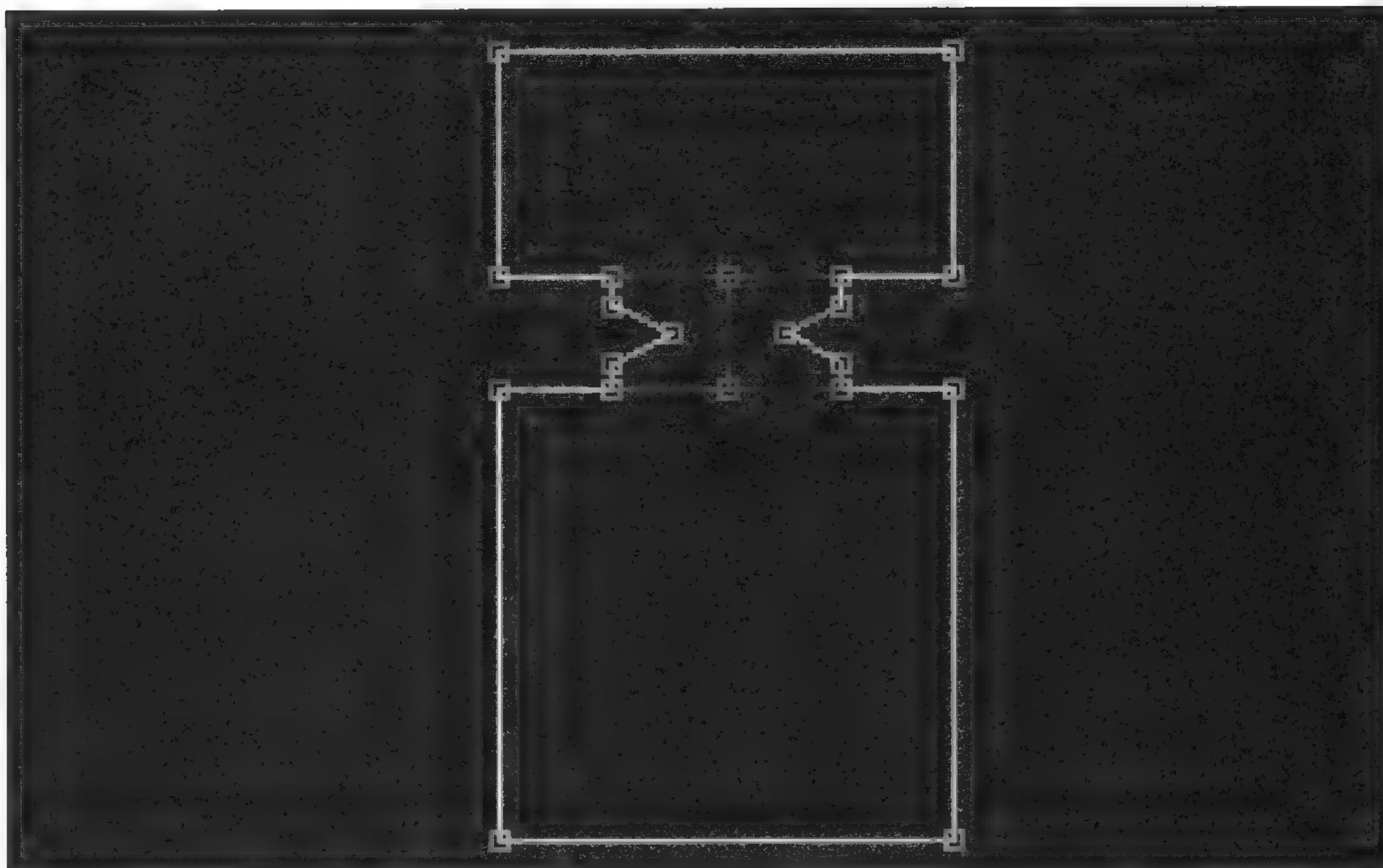


FIGURE 7.19: Create a slide door as you do a Star Trek door, except this time split the door sector in two.



NOTE

Interestingly, the *bug* mentioned previously where a door sound plays twice (for the Star Trek doors) does *not* occur for the slide door, so it may be preferable to use the slide door effect rather than the Star Trek door effect, even though this one is a bit harder to create.

point east, and the eastern door's sprite should point west. The final product should look similar to the one you see in Figure 7.20.

You can add a Music&SFX sprite to control the sound that plays when the door opens and closes, and a GPSSpeed sprite to change the speed of the door.

CREATING A SPLIT STAR TREK DOOR (ST 26)

Difficulty: Medium

To create a split Star Trek door, as shown in Figure 7.21, you must first create a Star Trek door. See the section “Star Trek Doors (ST 9).” Once you have a Star Trek door, change the door sector's LoTag value to 26. Finally, bring the ceiling and the floor of the door sector together so that they meet in the center of the door sector.

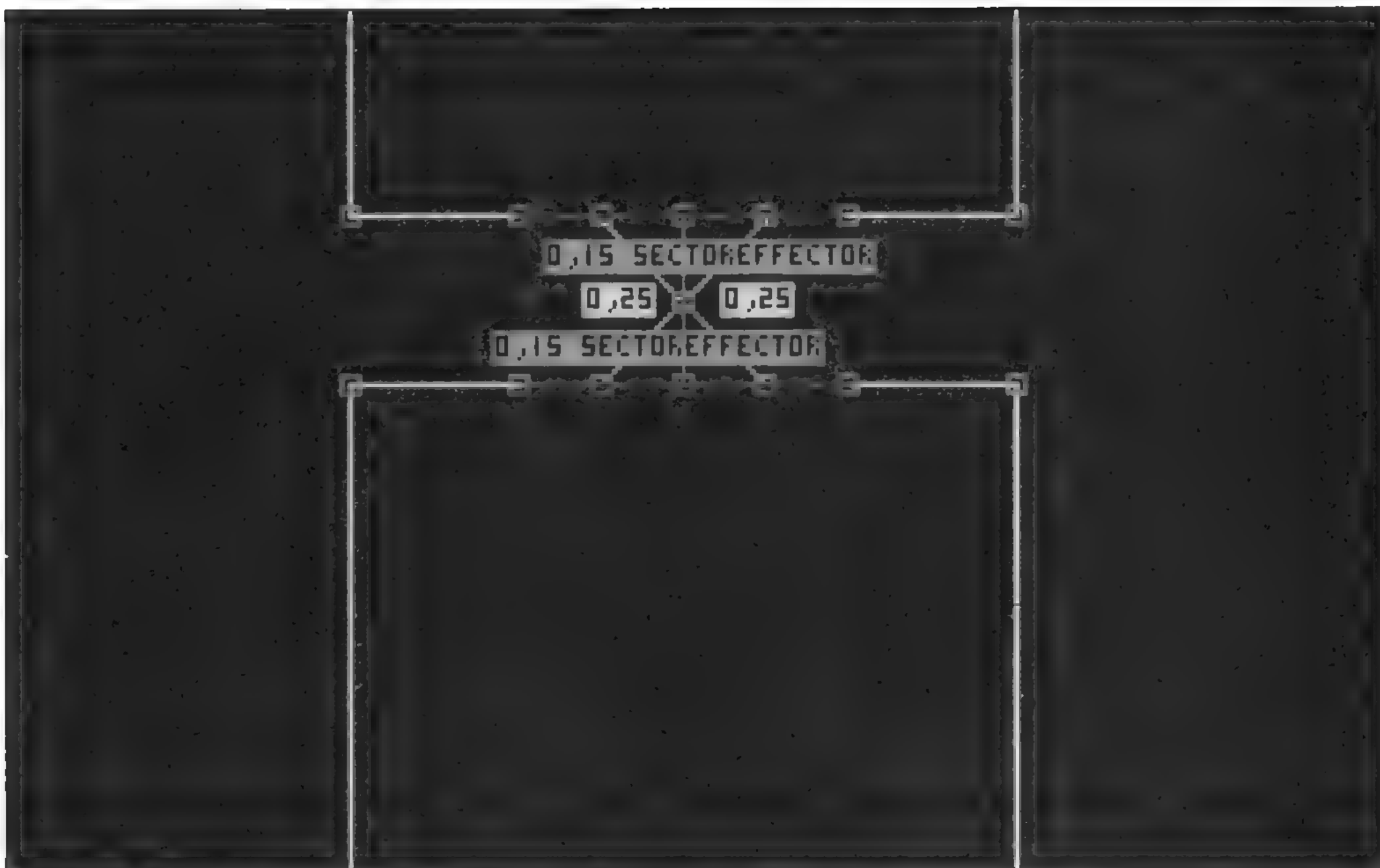


FIGURE 7.20: Here are the completed slide door sectors.

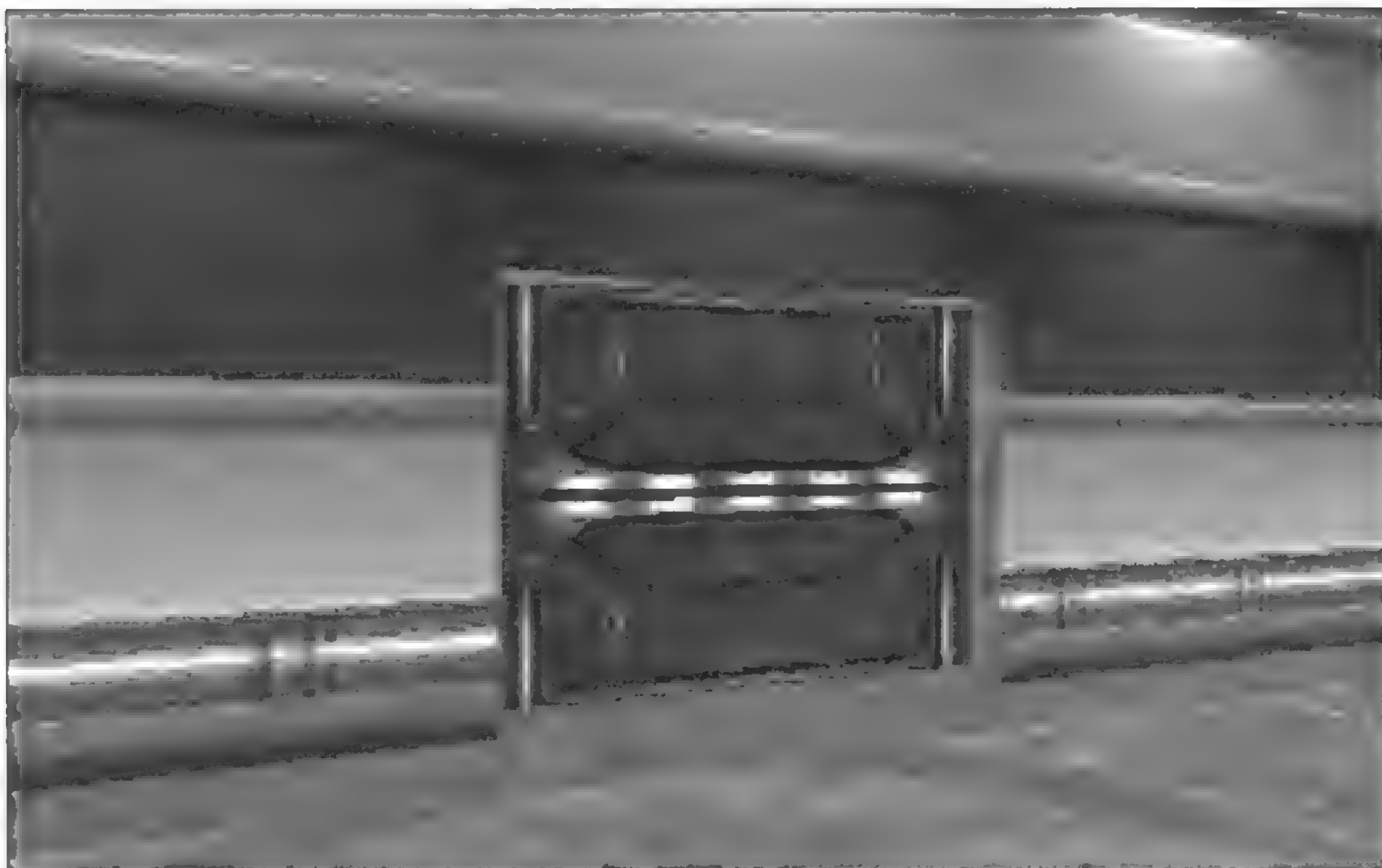


FIGURE 7.21: Here is a split Star Trek door in the closed position.

THE BRIDGE (ST 27)

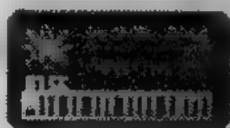
Difficulty: Medium

The Bridge effect (ST 27) combined with SectorEffector 20 controls bridges that



TIP

For some odd reason, the location of the SectorEffector sprite within the bridge sector seems to have an effect on the sector's movement. Make sure to center the sprite on the line that is going to extend. That is, if the sector is going to stretch south, make sure the SectorEffector is centered east to west in the bridge sector.



NOTE

The LoTag gets a unique value, not the HiTag, for the Activator (2) sprite.

extend into place when activated. Begin this effect by creating a large room sector. On one side, create a smaller, square child sector inside the room. This child sector will be the bridge. Raise the bridge sector's floor higher than the room sector's floor, and give this sector a LoTag value of 27.

Next, place a SectorEffector inside the bridge sector, and give it a LoTag value of 20. Change the angle of this sprite to point in the direction that you want the bridge to extend.

Add an Activator sprite (2) to the bridge sector. Give this sprite a unique LoTag value.

Finally, place one of the Switch sprites on a wall somewhere in the room sector (such as LIGHTSWITCH #134), and give it the same unique LoTag value that you gave the Activator sprite. That connects the flipping of the switch to the Activator sprite. When the Activator sprite is activated, it

triggers all SectorEffectors in the same sector. In this case, SE 20 will be triggered to extend the bridge. That should complete your bridge. Go and try it out in the game.

Figure 7.22 shows an example of two bridges that extend toward one another. The only difference between the two bridges in the figure is the speed at which they extend, which is controlled by the LoTag values on the GPSSpeed sprites.

DROP FLOOR (ST 28)

Difficulty: Medium

To create an effect that changes the height of a floor or ceiling, use ST 28, SectorEffector 21, and a MasterSwitch or Activator sprite. Follow the directions here to create this effect, as the instructions for it in the Build documentation are in error.

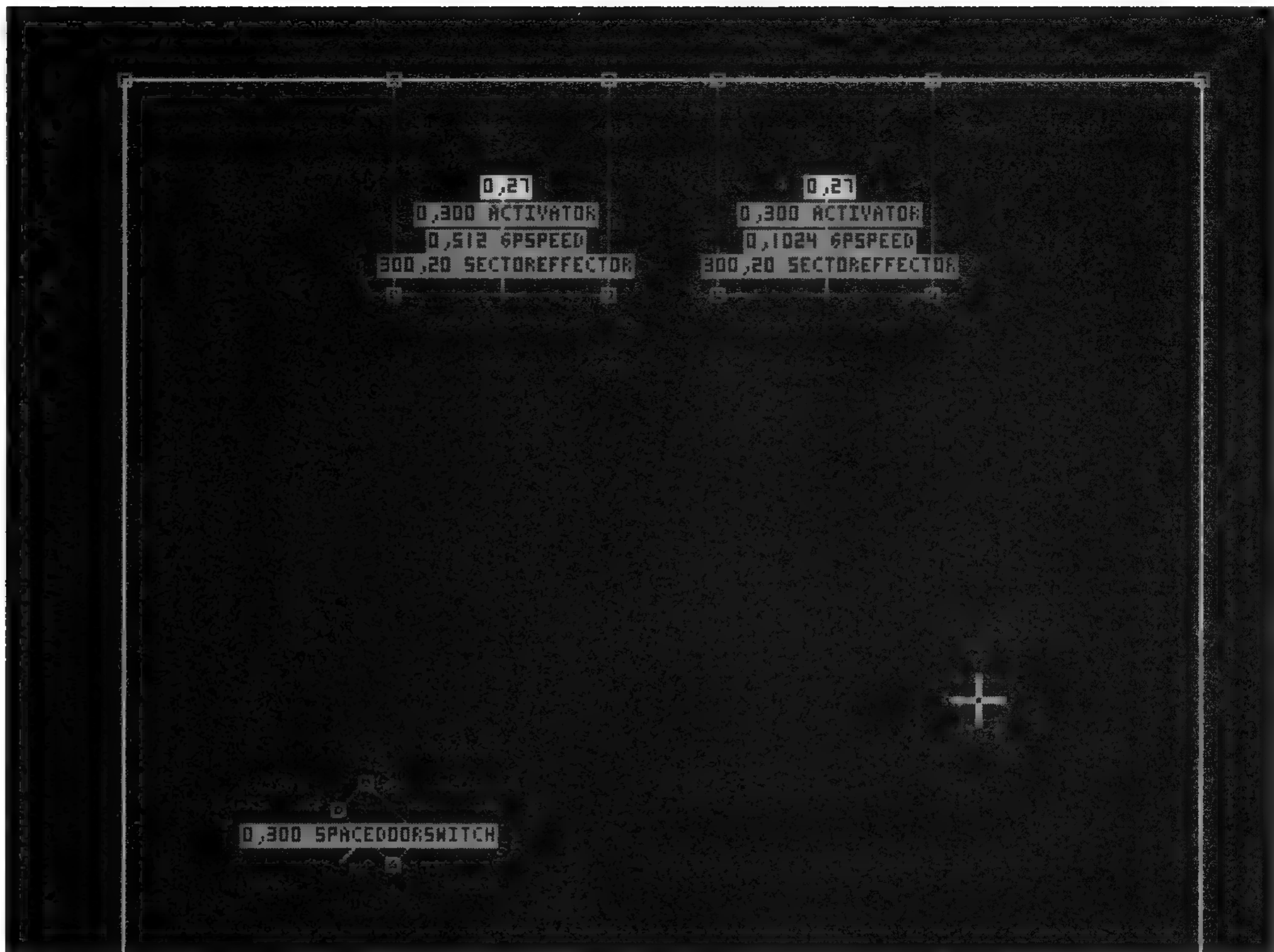


FIGURE 7.22: In this bridge example from the level map stored in `_ST.MAP`, two bridges extend when the player flips the `SPACEDOORSWITCH` sprite.

Make a sector for a room in which you want the ceiling or floor to lower, and create an adjoining sector next to it, from which you can watch the action. In the sector to have the drop floor (or ceiling), design the ceiling and floor heights the way that you want them to be *before* the drop action occurs. Assign a LoTag value of 28 for this sector.

Add a SectorEffector sprite to the sector with the drop floor (or ceiling). Give it a LoTag value of 21. If you want the ceiling of your sector to drop, make the angle of this sprite point upward (north). If you want the floor to drop, point the angle of the sprite downward (south). Finally, set the height of the SectorEffector sprite to be the height you want the ceiling or floor to drop. You may have to lower the sprite below the current level of the floor to accomplish this.

Next, add an Activator sprite to the sector with the drop floor (or ceiling), and give it a unique LoTag value. Finally, add a child sector to the adjoining sector from where you'll watch the action. Add a Switch sprite (such as 713) within this child sector. Give

the Switch sprite the same LoTag value as you gave the Activator sprite. That completes the dropping floor or ceiling effect. Go ahead and try it out in the game.



QUICK FIXES

The Build documentation tells you to design the sector heights the way they should look *after* the drop action and to put the SectorEffector at the start point. This is the opposite of what you did here.

Figure 7.23 provides an example of Sector Tag 28 from the map stored in the file `_ST.MAP`. Notice that four sectors are moved when the switch in the southwest corner is flipped, and each moves differently to present all four combinations of the way ceilings and floors can move up and down. Figure 7.24 shows a 3D view of this example.

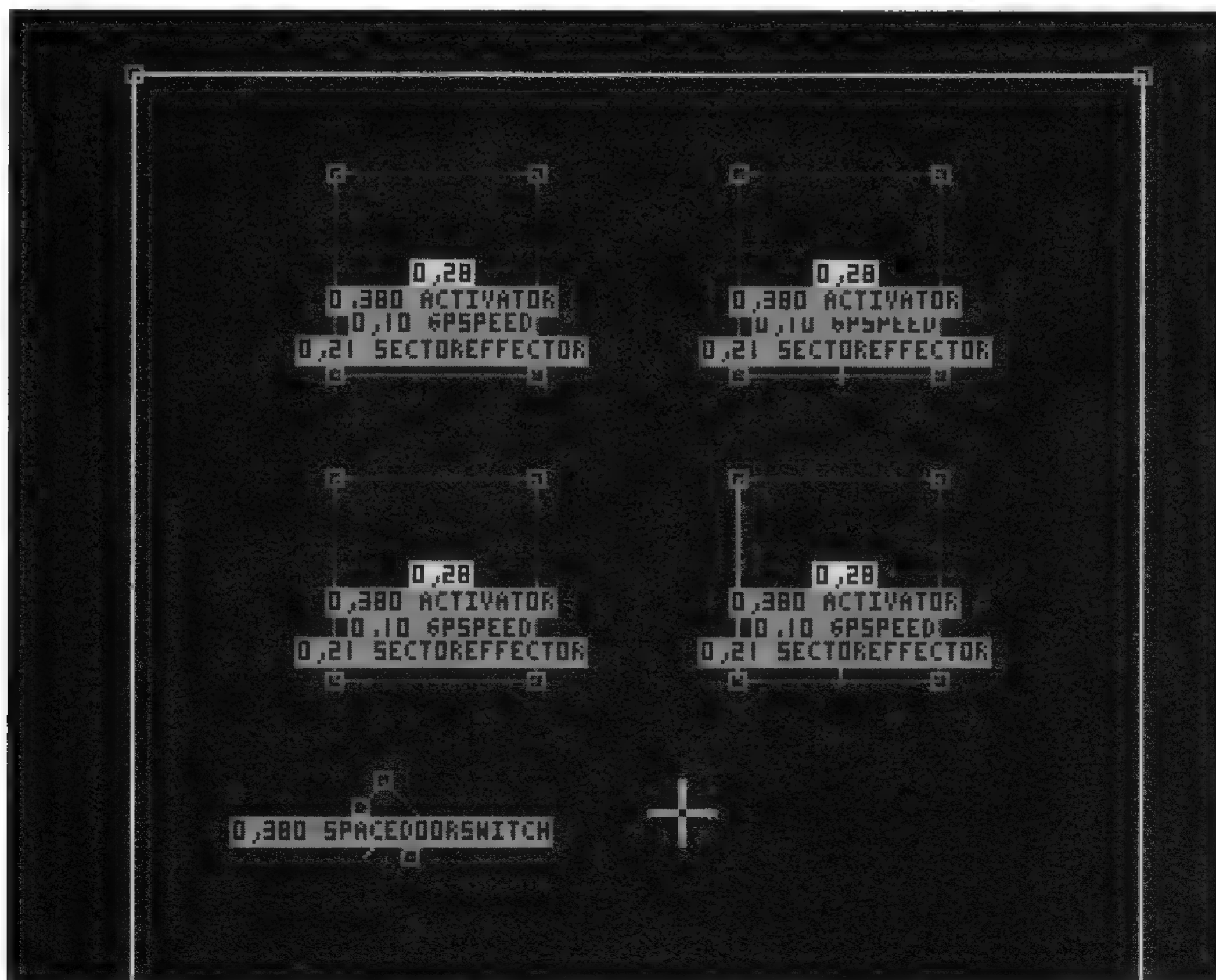


FIGURE 7.23: This example of a ST 28 effect can be found in the file `_ST.MAP`.



FIGURE 7.24: In this 3D view of the same ST 28 example, notice that the heights of the SectorEffectors are instrumental in defining the movement for each sector.

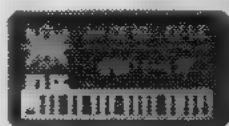
TEETH DOOR (ST 29)

Difficulty: Medium to Hard

The teeth door effect provides a door with extra reinforcing bars that descend from the door's bottom and appear to sink into the ground as the door closes. Begin this effect by making the familiar arrangement of two sectors for adjoining rooms with an intervening sector representing the door. This time, however, make the door sector wider, so you have room to add the teeth.

Add any number of child sectors to the inside of the door sector. Make sure to press Alt + S inside each child sector to make all the lines of these sectors red (two-sided), which makes the child sectors valid player spaces. These child sectors will become the *teeth*. Then, lower the floor of each of the child (tooth) sectors so that each floor is well below the floor level of the door sector. This causes the teeth to descend faster than the door when it closes so that the player will see the teeth fall into place.

Add a SectorEffector sprite to each of the teeth sectors and to the main door sector. Give each of the SectorEffector sprites a LoTag value of 22 and a unique HiTag

**NOTE**

The Build documentation mentions that this door was not tested very well and that it may crash. If it does, you may have to avoid using it; however, I haven't seen a problem so far in my testing of this effect.

value that doesn't appear elsewhere on your map. Give the main door sector itself a LoTag value of 29 and the same unique HiTag value that you gave the SectorEffectors. The final product should look like the one shown in Figure 7.25.

You can add a Music&SFX sprite to control the sound that plays when the door opens or closes and a GPSSpeed sprite to change the speed of the door. Go ahead and try out your teeth door effect in the game.

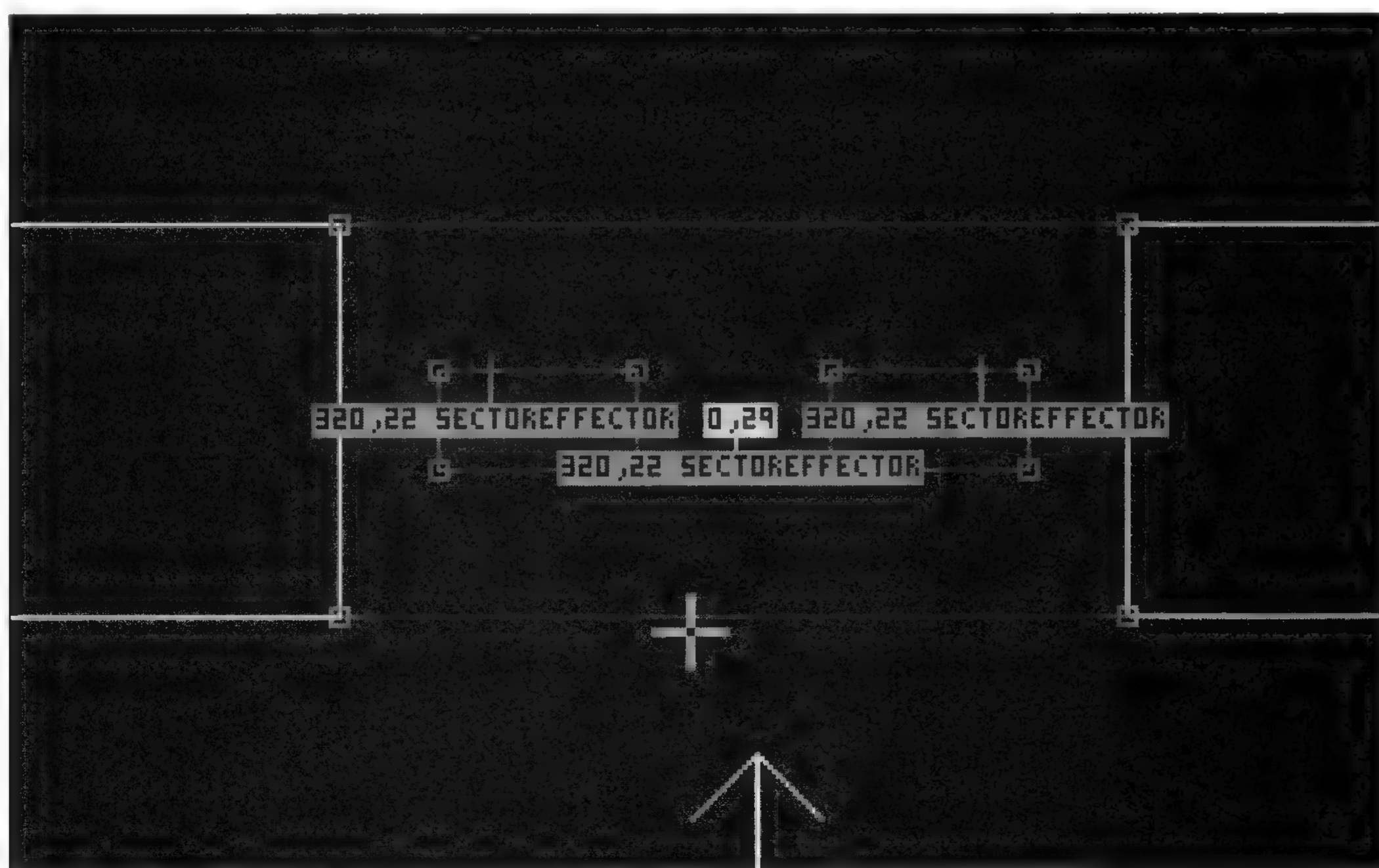


FIGURE 7.25: Here's the final layout for the teeth door sectors.

ROTATE RISE DOOR (ST 30)

Difficulty: Hard

The Build documentation is a bit unclear on the name of this sector effect. In one place it's named "rotate rise bridge," and in another place it's named "rotate rise door."

In any case, this highly specialized sector rotates around a point, while its floor rises. The example room in the file `_SE.MAP` shows an excellent example of this sector.

To make this type of sector, create a room with a child sector of any shape, away from the center of the room. Give the child sector a LoTag value of 30. Raise the floor of the child sector a bit, so that you can differentiate it from the parent room. Then, place a SectorEffector sprite in the child sector, assign it a LoTag value of 0 (rotating sector), and a unique HiTag value. The height of this sprite will be the height that the sector rises when activated, so raise the sprite well above the floor level in 3D mode.

Now, place a second SectorEffector somewhere in the main room to act as the rotation point for the rotating sector. Give this sprite a LoTag value of 1 and the same unique HiTag value as you gave the SectorEffector 0 sprite above. Put an Activator sprite (2) in the rotating sector, and give it the same unique value for its LoTag as you used for the SectorEffector's HiTag value.

All you need to do to complete this effect is place a Switch sprite somewhere in the room on a wall, and give it the unique value for its LoTag that you assigned for the Activator's LoTag value to link the two. That should complete the effect. You can go try it out in the game. Figure 7.26 shows an example of this effect, which can be found in the file `_ST.MAP`. It consists of three sectors that all rotate around a common point when the switch is flipped.

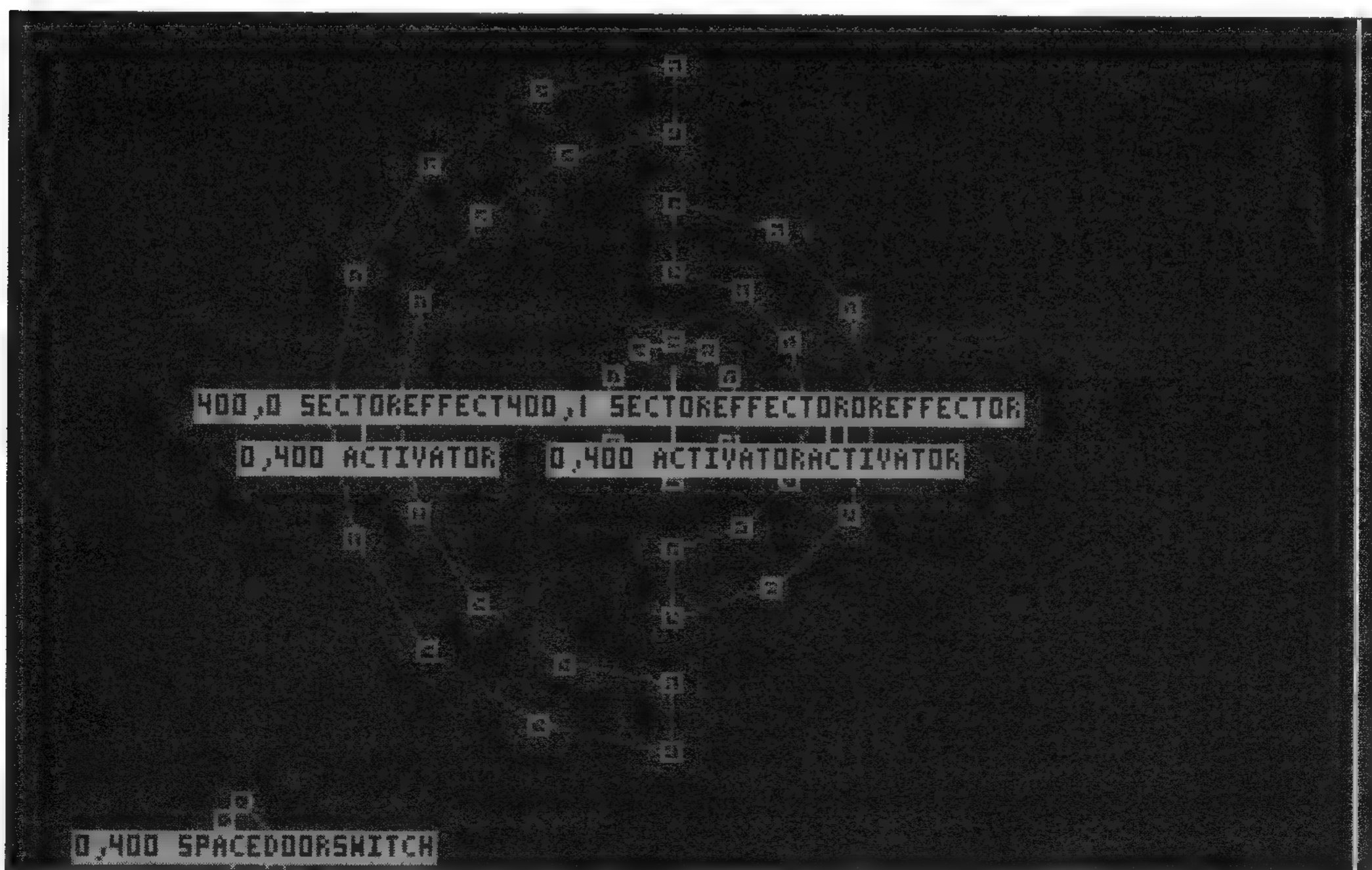


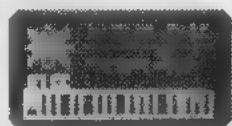
FIGURE 7.26: This is an example of the rotate rise door effect found in the file `_ST.MAP`.

TWO-WAY TRAIN (ST 31)

Difficulty: Medium to Hard

When I looked at the example for this effect in the file `_SE.MAP`, I thought it was going to be very hard to make because of all the different SectorEffectors and Switch sprites. However, if you reduce the two-way train to its basic elements, it's no harder to create than any other sector effect.

Start with a long sector to act as the track for the train. Make it wide enough to hold a child sector. Add two Locator sprites (6) to the opposite ends of the long sector. These define the points that the train will pass between. Give one Locator sprite a LoTag value of 1, and leave the other at 0.



NOTE

In this example and some others that follow, I use the generic letter **x** to signify a unique LoTag or HiTag value that you should choose to link the necessary sectors and sprites needed to complete the effect. Choose any number that you haven't used on your map to this point, and use that number in place of **x**.

Design a child sector on one end of the long parent sector. Raise the floor of this sector. This child sector will be the train. Assign it a LoTag value of 31. Place an Activator sprite in the train sector. Give it a unique LoTag value, say **x** for the example. Also place a SectorEffector sprite in the train sector. Give it a LoTag value of 30, and a HiTag value that is the same number you assigned the Activator sprite's LoTag value (**x**). Make the angle of the SectorEffector sprite point toward the front of the car. In your example, the angle would point down the long sector toward the far Locator sprite.

Then, place a Switch sprite somewhere on the wall of the long sector. Give its LoTag the

same value you gave the Activator sprite's LoTag and the SectorEffector's sprite's HiTag (**x**) to tie it to the Activator sprite.

This completes the minimum requirements for the two-way train, which is shown in 2D view mode in Figure 7.27 and in 3D view mode in Figure 7.28. You can go ahead and try it out in the game.

You can further enhance the two-way train with additional sector effects nearby. When the train stops at Locator A, any Activator sprite on your map with a LoTag value of $x + 1$ will automatically activate. When the train stops at Locator B, any Activator sprite with a LoTag value of $x + 2$ will activate. If you don't plan to use this feature for your train, make sure to skip unique LoTags $x + 1$ and $x + 2$ on your map, or your train may activate some sector effect clear across the level!



FIGURE 7.27: Here are the two-way train sectors shown in 2D view. The angle of the SectorEffector sprite can't be seen in this picture; it is heading due east to denote the front of the train.

By using these extra Activator sprites, however, you can cause a door to automatically open when the train stops. This is used to great effect on the Warp Factor level in Lunar Apocalypse (E2L3), where the *train* is made to look like a small shuttle pod in space, and airlocks automatically open when the pod docks.

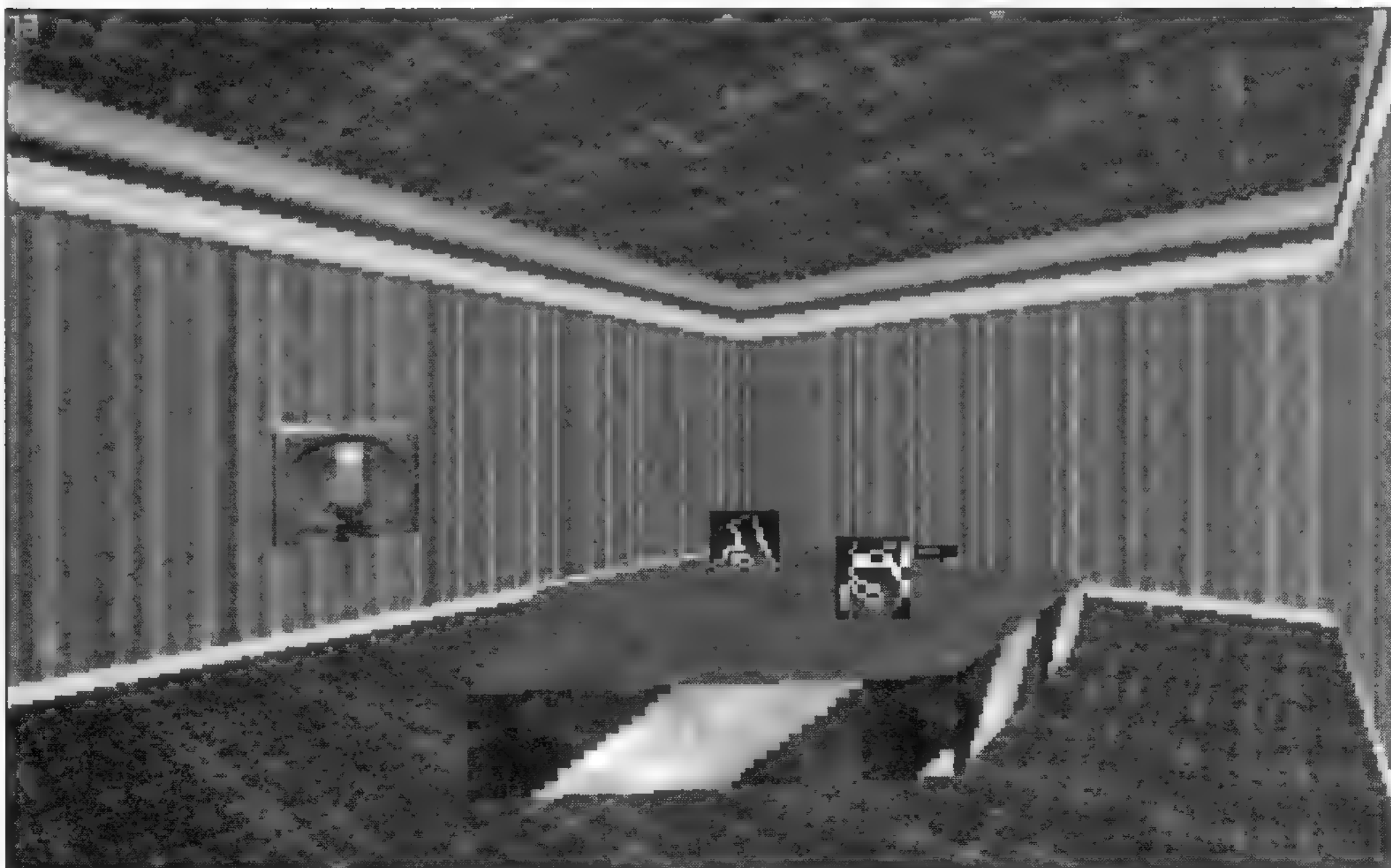
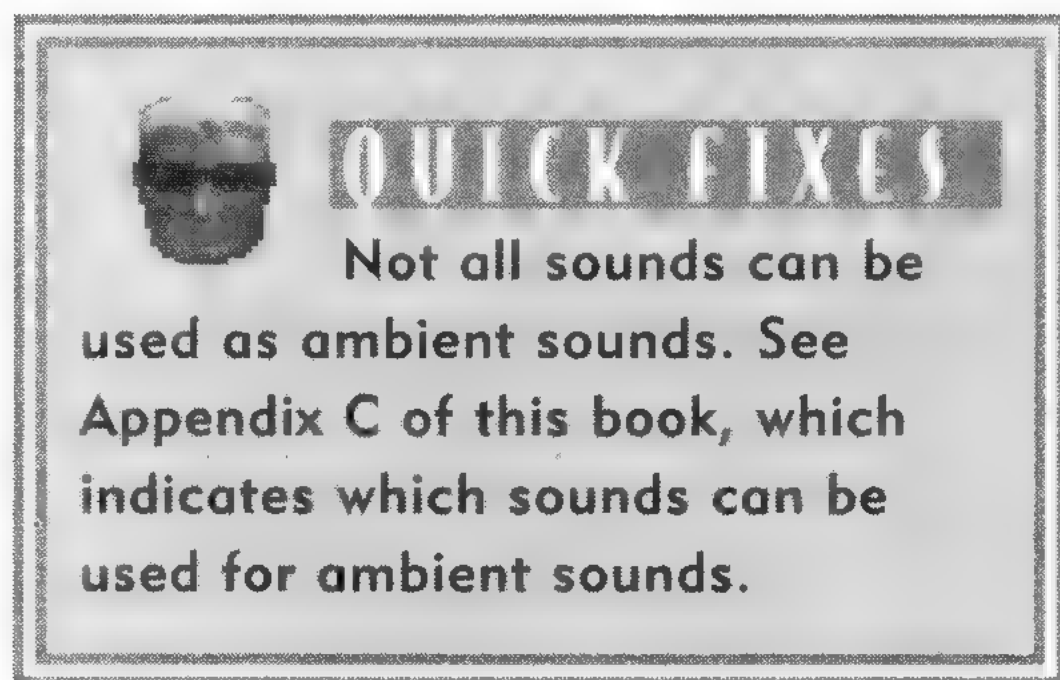


FIGURE 7.28: Here is the two-way train effect shown in 3D view.

APPLYING AN AMBIENT SOUND (ST 10000+)

Difficulty: Easy

If you would like to play a sound when a player enters a sector for the first time, first find the sound number you want in the DEFS.CON file. Add 10000 to the number listed for the sound, and assign the result as the LoTag value for the sector. The sound will play one time when the player steps into the sector. This can be used for ambient effects as the player enters a new area, which can be quite effective in setting mood.



SECRET AREAS (ST 32767)

Difficulty: Easy

A sector with a LoTag value of 32767 will be flagged as secret, and it will count toward the "Secrets Found/Secrets Missed" message displayed at the end of the level. Make sure to include a few of these sectors in your levels, and reward the player for finding them by stocking them with goodies such as weapons, ammo, and power-ups.

ENDING THE LEVEL (ST 65535)

Difficulty: Easy

You can give a sector a LoTag value of 65535, and the level will end automatically when Duke walks into this sector. Sectors marked in this way are obviously made to appear like some type of exit so that the player doesn't just blunder onto the sector and end the level unexpectedly.

You can also use the NukeButton sprite (142) to end the level or to go to a bonus level. To use this sprite, set the LoTag value to 32767, if you want the user to go to the next level, or set it to the number of the next level (1–11) to go to a bonus level. If you do want to use the NukeButton sprite to go to a bonus level, also set the palette of the sprite to 14.

USING SECTOREFFECTORS

All of the special sector effects discussed thus far have required the use of a Sector Tag with a LoTag value to help define them. In some cases, the Sector Tag also requires a matching SectorEffector to help create the effect. In other cases, the sector's LoTag value alone defines the effect.

This next section will cover another category of effects—those that do *not* require any type of sector LoTag value. Instead, these effects will be created primarily with some type of SectorEffector, and possibly some other sprites. As in the section covering Sector Tags, the SectorEffectors will be listed numerically along with a relative difficulty rating, so you can judge for yourself which you might want to practice first.

In many cases, certain types of SectorEffectors are used in conjunction with the Sector Tags described previously. For these SectorEffector types, I will simply refer you to the appropriate Sector Tag section.

ROTATING SECTOR (SE 0)

Difficulty: Medium

Rotating sectors are used for the giant spinning gears that you see in the game and for many other effects. A sector can rotate around any point. The point does not necessarily have to lie inside the rotating sector; you can have a sector *orbiting* around a point somewhere else in the room.

To use this SectorEffector, begin by drawing a rectangular sector. This will be the *container* sector for the rotating sector. Draw a child sector somewhere in the first sector, but not in the center. Press Alt + S to make it a child sector. Then, switch to 3D view mode, and move the floor of the child sector up so that it's above the level of the parent floor. Also, press the R key while pointing at the floor and ceiling to make the textures move with the sector.

Place a SectorEffector sprite inside the child sector. Because the SectorEffector type for a rotated sector is 0, you don't have to edit this sprite's LoTag value—leave it as 0. Then, give this sprite a unique and unused HiTag value. Place a second SectorEffector sprite in the center of the room. Give this sprite a LoTag value of 1,



TIP

Recall that references to the type of SectorEffector refer to the LoTag value that you assign the SectorEffector. For example, a "SectorEffector 3" is simply a SectorEffector sprite with a LoTag value of 3.

which makes the sprite act as a pivot point. Give it the same unique HiTag value that you gave the first SectorEffector. If the angle of this sprite faces up, the sector will rotate clockwise; if the angle faces down, the sector will rotate counterclockwise. Change the angle of this sprite as you desire. The final result should look like what you see in Figure 7.29.

You can make more than one sector rotate around the same pivot point by adding a SectorEffector 0 to each rotating sector and giving it the same HiTag value you gave the pivot SectorEffector. You can also add a Music&SFX sprite to the rotating sector to define an ambient sound.

By the way, don't let the paths of rotation for rotating sectors cross outside of their

parent sector; Build doesn't draw the effect correctly when you do this. Your sector should now rotate around SE 1. Go check it out in the game.

To make a sector spin around its own center, like the giant gears on the Death Row level (E1L3), simply follow the directions for the

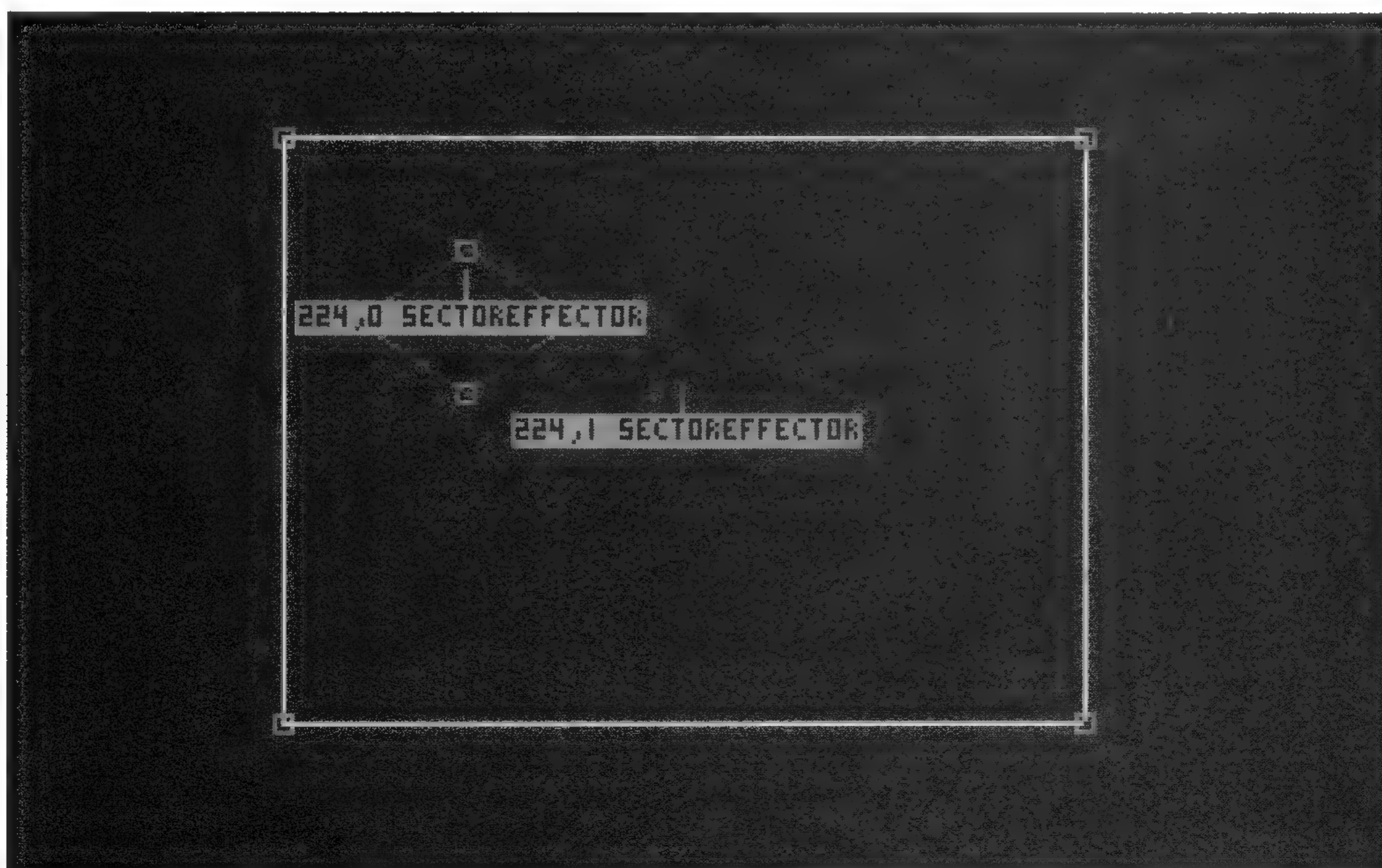
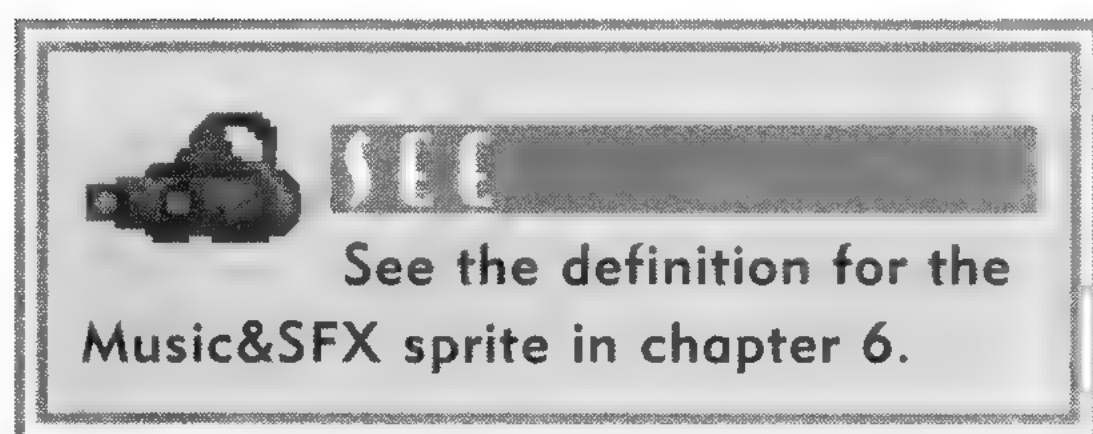


FIGURE 7.29: Here is a 2D view of a rotating sector.

rotating sector (above), and put the pivot point (SE 1) at the exact center of the sector you want to rotate.

PIVOT POINT FOR A ROTATING SECTOR (SE 1)

Difficulty: Medium

The use of this SectorEffector is covered in the preceding section, which describes creating a rotating sector with SE 0.

EARTHQUAKE (SE 2)

Difficulty: Medium to Hard

Earthquakes are a little more complex than some of the other special effects. Follow along here, and you'll have the place shaking in no time.

First, draw a large room to house the earthquake. Then, create a child sector within the parent. Make the child sector an irregular shape by adding vertices and moving them around. You also want to tilt the floor of the child sector by using the [and] keys in 3D view mode. Remember you can also change the direction of the tilt by changing the first wall of the sector (Alt + F). Make the sector look like you want it to look *after* the quake hits.

Now, place a SectorEffector sprite in the earthquake (child) sector and give it a LoTag value of 2 to make this an earthquake sector. Also, change the angle of this sprite to face the direction you would like the sector to move during the earthquake. Make sure this sprite is sitting on the floor level *before* the quake, that is, the floor level of the parent sector.

Next, add a MasterSwitch sprite (8) to the earthquake sector, and raise it up to the same height as the SectorEffector sprite. Give this sprite a unique LoTag value. Notice I said *LoTag* here for the unique value, which is different from most other sector effects. You can also give the MasterSwitch sprite a HiTag value; the HiTag value will be the amount of the delay before the earthquake is triggered. Leaving the HiTag value at 0 results in the earthquake happening immediately.

The next step is to build another child sector nearby that will act as the triggering mechanism for the quake when the player stands on it. You may want to either raise the floor of this sector a bit or give it another texture so you can find it when you play the game (see Figure 7.30). Sectors that trigger an action when stood upon are called *touchplates*. They are created by using a TouchPlate sprite. Add a TouchPlate

sprite (3) to this new triggering sector, and give it the same LoTag value that you gave the MasterSwitch sprite, as shown in Figure 7.31.

As an additional feature, if you would like falling debris during your quake, add some SectorEffector 33 sprites in the area. These sprites automatically spawn earthquake *jibs*, or debris. If you want the debris to fall from above, put these sprites on the ceiling of the sector. If you want them to come from the ground, leave these sprites at floor level. You don't have to match up any LoTags or HiTags for this SectorEffector. Earthquake jibs will be created from these sprites anytime an earthquake is triggered anywhere on the map.

Your earthquake should be fully operational, so go check it out in the game.

RANDOM LIGHTS AFTER SHOT OUT (SE 3)

Difficulty: Medium to Hard

One great lighting effect that you can create is to set up lights that can be shot out, causing a blinking effect for a few seconds before the appropriate sectors go dark. This is done using SectorEffector 3. The lights to be shot out can be placed either on the

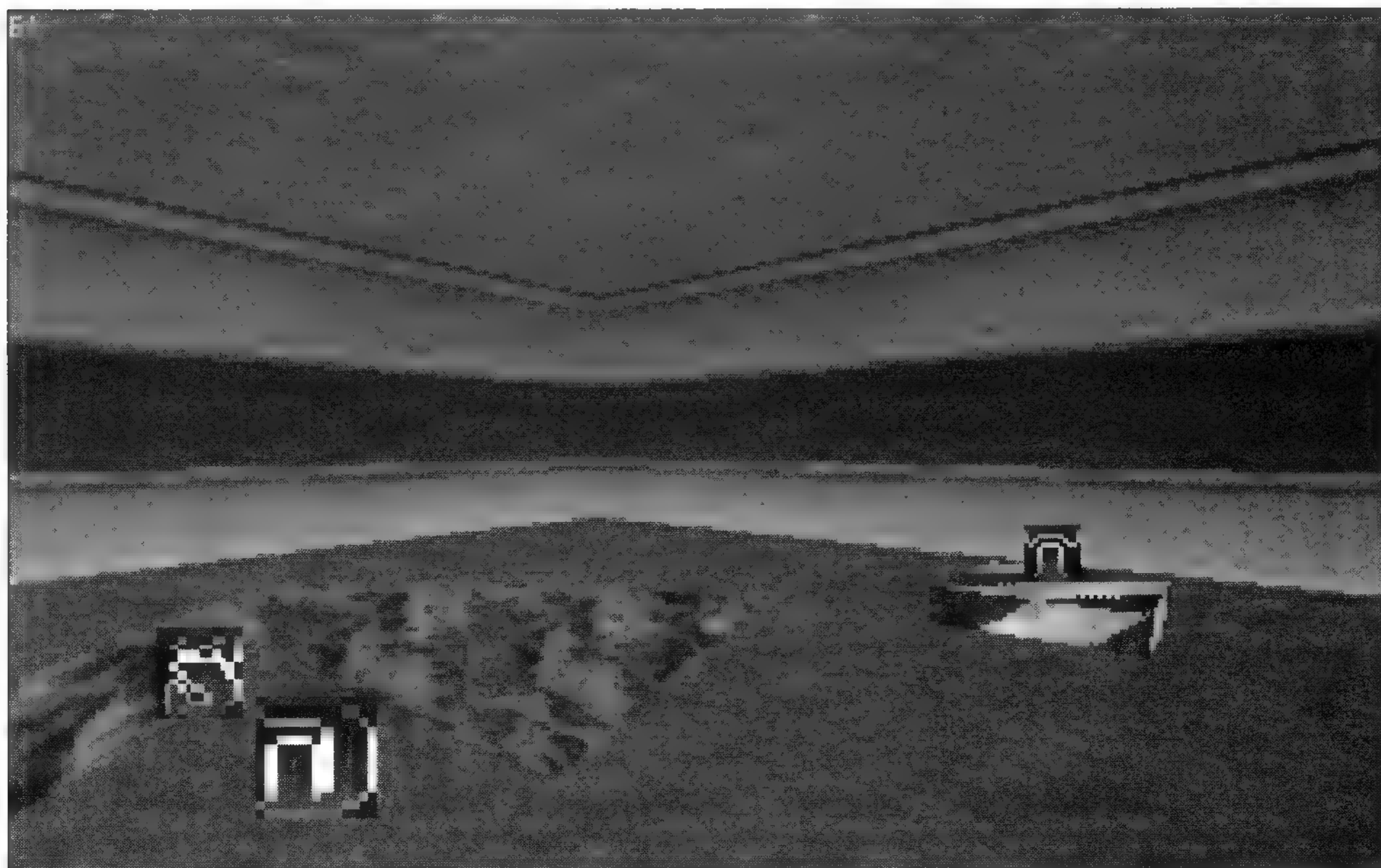


FIGURE 7.30: This 3D view of the example earthquake sectors shows raised floor heights for the SectorEffector and MasterSwitch sprites, so they are distinguished from the parent sector.

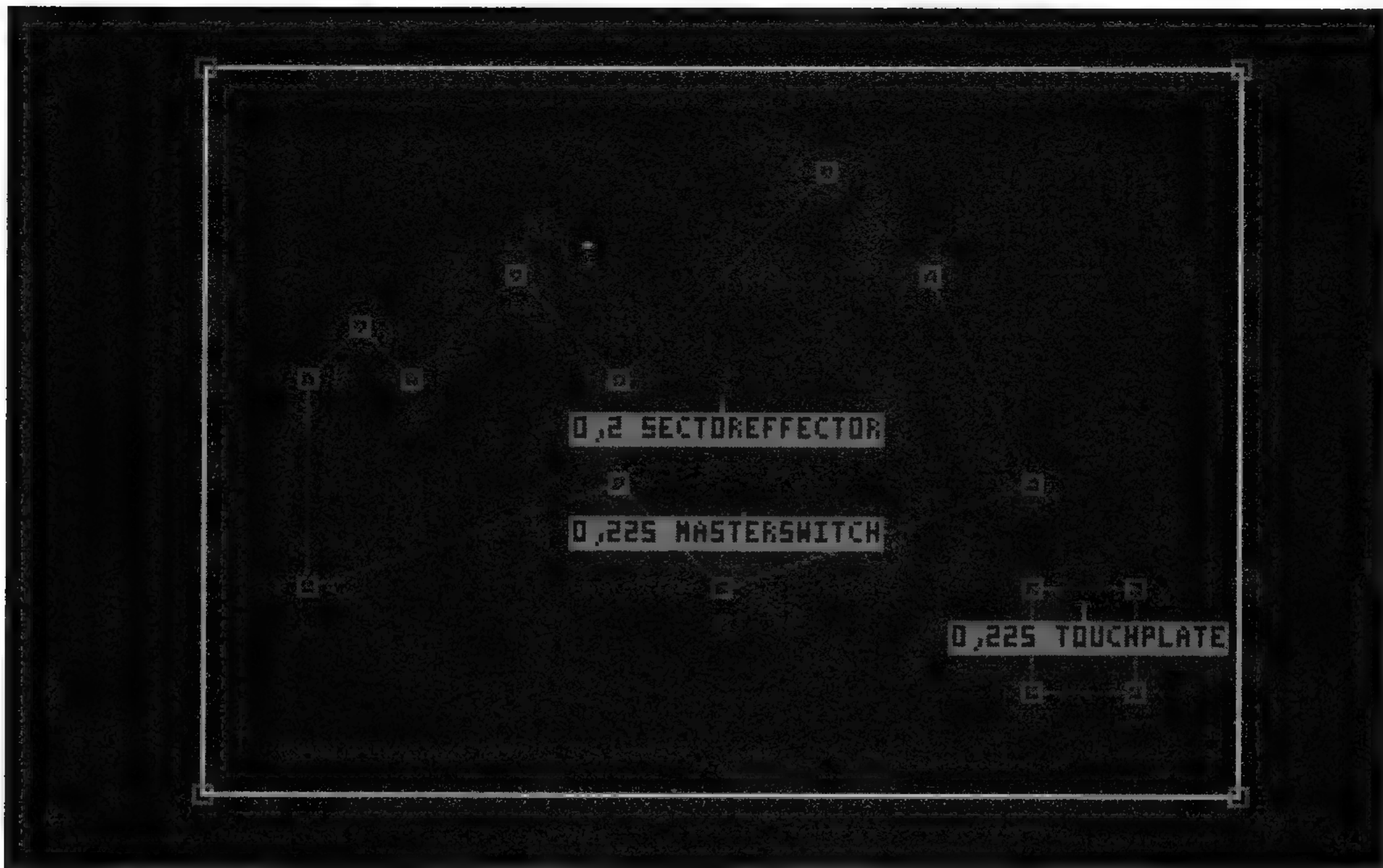


FIGURE 7.31: Set up your earthquake sector like this.

ceiling or on the walls, but the technique for each effect is different, so I will explain both here. Let's start with the lights on the ceiling, as they are a bit easier to create.

Ceiling Lights Effect

Start off by making a simple room and placing a child sector inside it. Give the ceiling of the child sector texture #120, TECHLIGHT2. You then may want to change the size of the sector and the scale of the ceiling texture to make the light's boundaries end on the sector boundaries.

Next, give the child sector a unique unused HiTag value (x). Then, darken the shade of the ceiling and floor of the sector to the shade you want them to be *after* the lights are shot out. Add a SectorEffector sprite to the child sector. Give the SectorEffector a LoTag value of 3, and a HiTag value of x to match the HiTag value of the sector. Finally, change the shade of the SectorEffector sprite to the shade you want the sector to be before the lights are shot out.

Note that you can create multiple sectors that are affected by the same lights by adding an SE 3 to each sector you'd like affected and giving each the matching HiTag value that you gave the first SectorEffector 3 (x). Make sure to shade both the floors and the ceilings of each sector, as well as each SE 3 sprite. Your final area should look

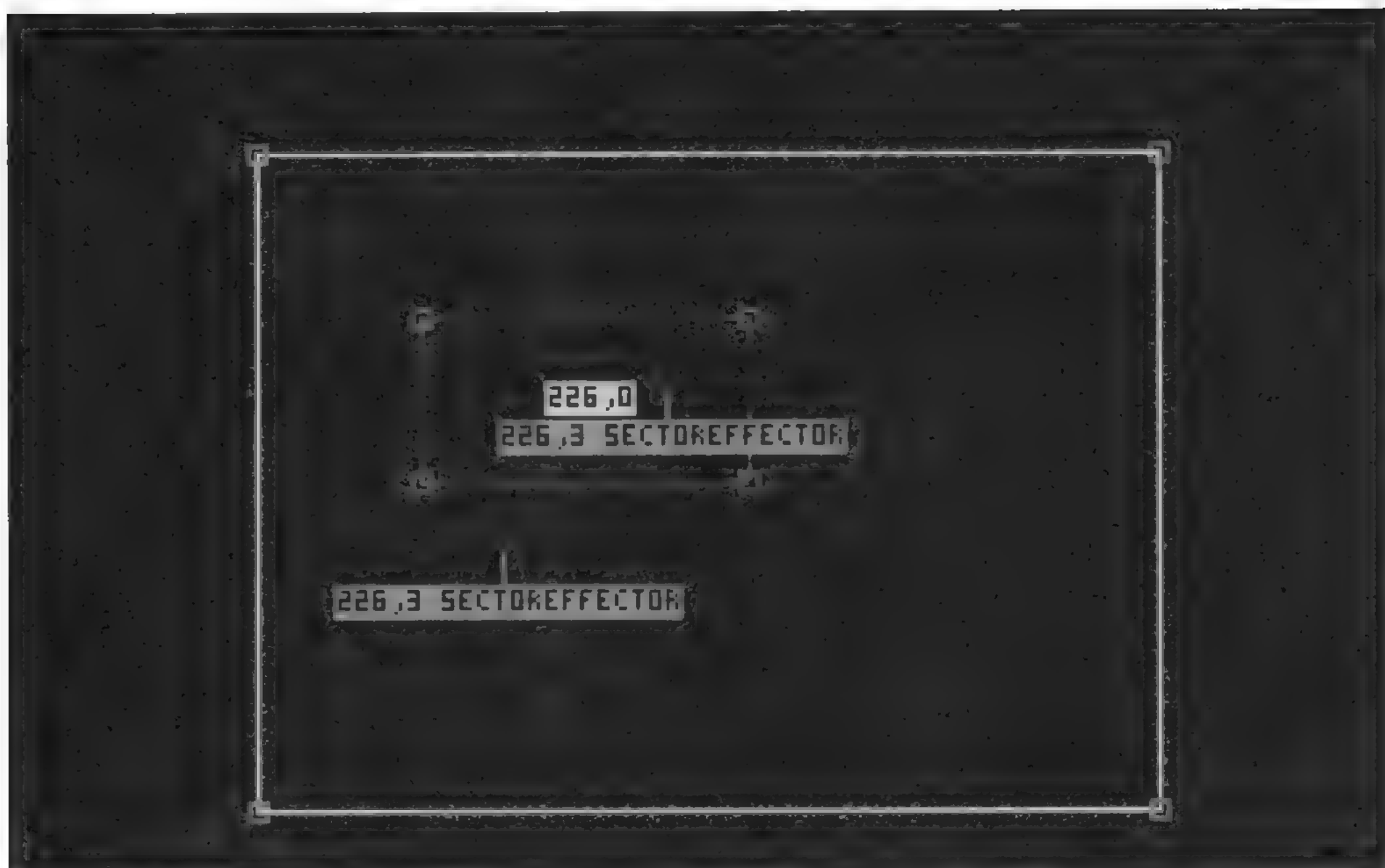


FIGURE 7.32: This is a 2D view of sectors that set up ceiling lights that will darken a sector or sectors after they are shot out.

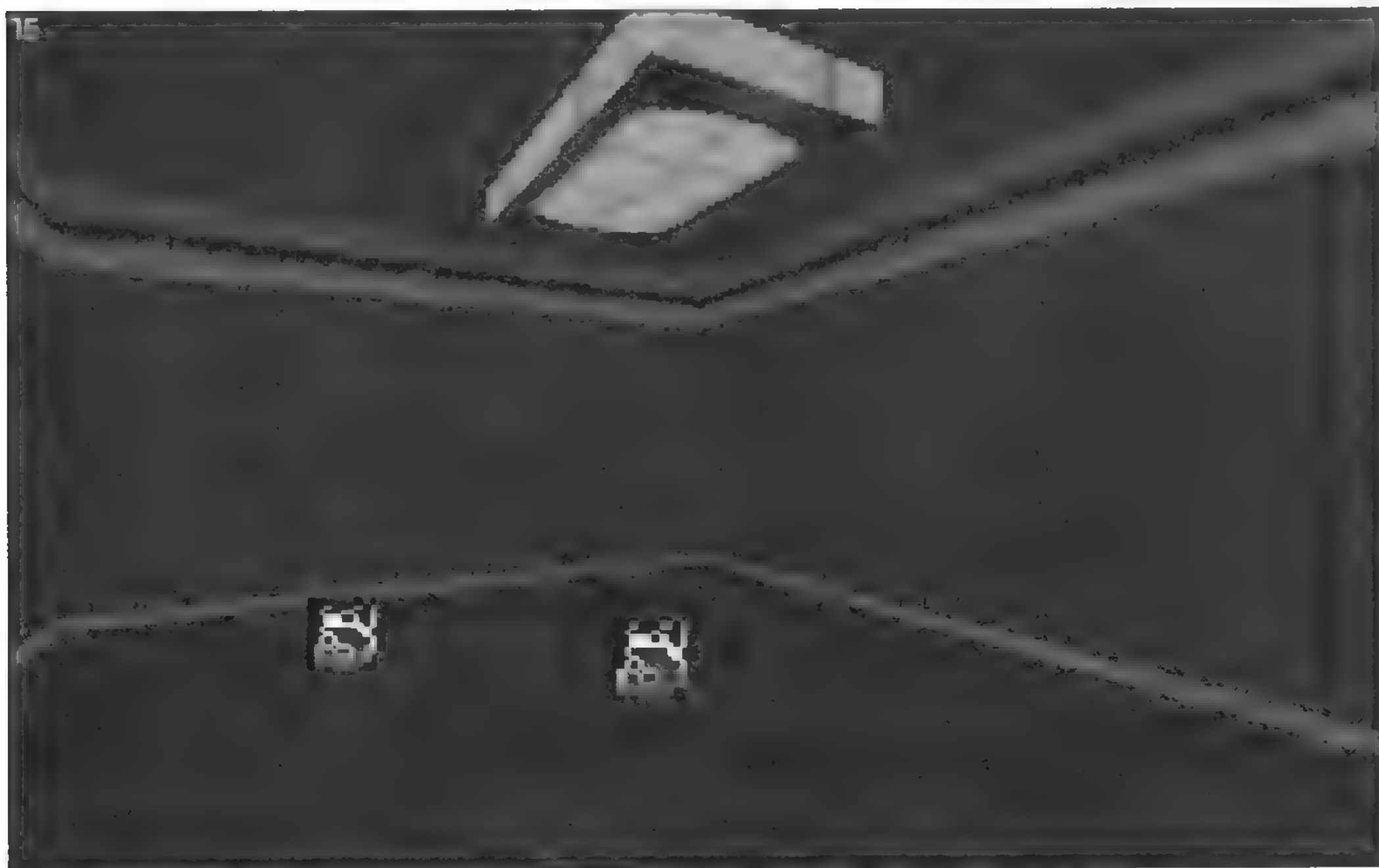


FIGURE 7.33: In this 3D view of the SE 3 setup, the shade of each wall, the floor, and the ceiling is the shade the sector should be *after* the lights are shot out.

and the ceilings of each sector, as well as each SE 3 sprite. Your final area should look similar to the one shown in 2D view in Figure 7.32 and in 3D view in Figure 7.33.

Wall Lights Effect

The steps are substantially different for creating the same type of effect based on a wall light being shot out. For these kinds of lights, start with an ordinary room, and split one of the walls of the room into three parts. Make a small sector off the main room. Then, split the main room sector by extending red lines from the original split wall outward to somewhere else along the room walls. See Figure 7.34 to get an idea of how to shape the sectors. Notice that the main room is divided into three sectors, the center of which will represent the main beam emanating from the wall lights, which are on the south end of the room.

Next, take the small triangular sector you made to the south of the room and lower its ceiling down to the floor. Give the upper part of the now-visible wall texture #120, TECHLIGHT2. Scale and pan the texture so that it looks like a realistic light on a wall.

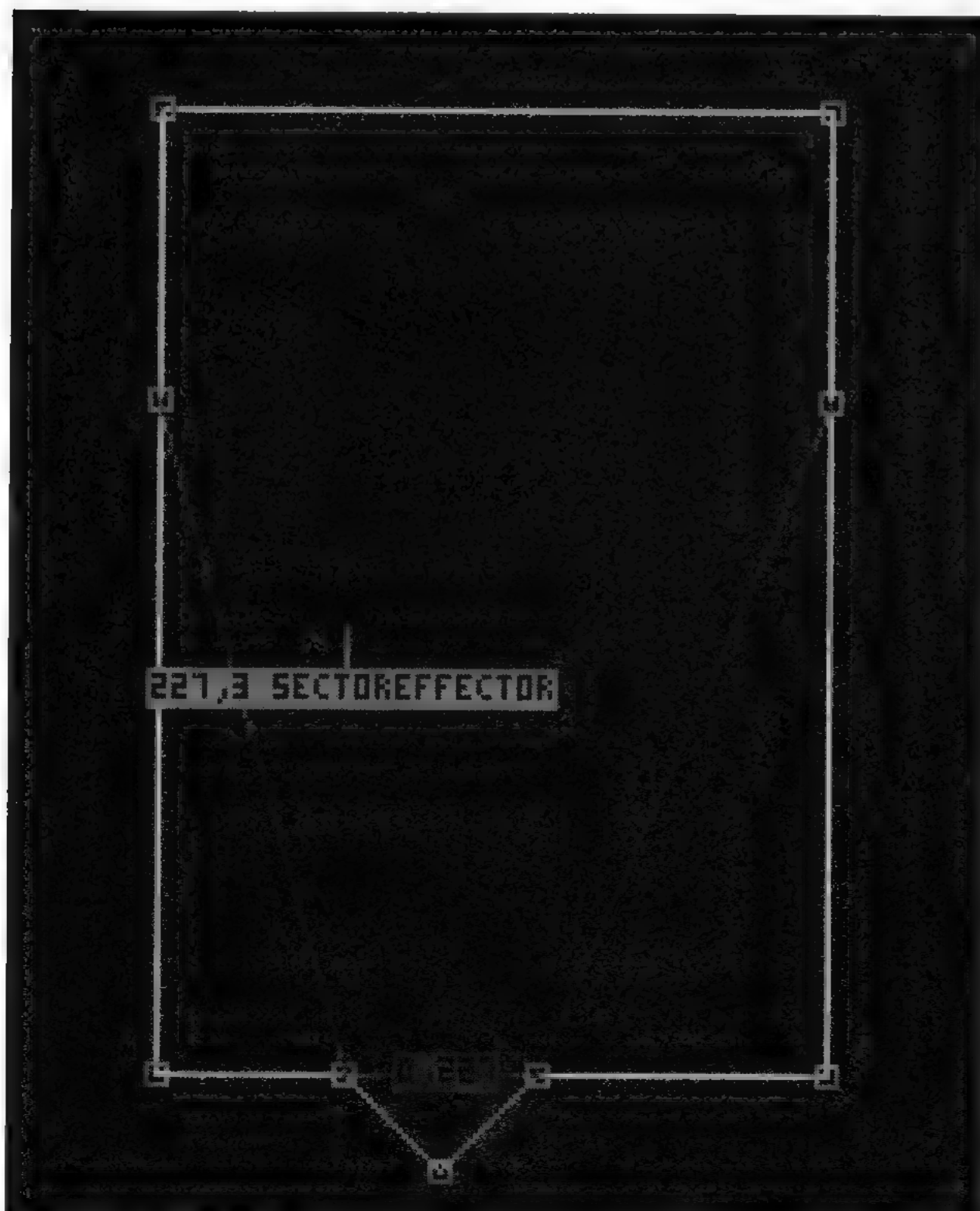


FIGURE 7.34: This level map shows sectors set up to create rooms that darken when wall lights are shot out.

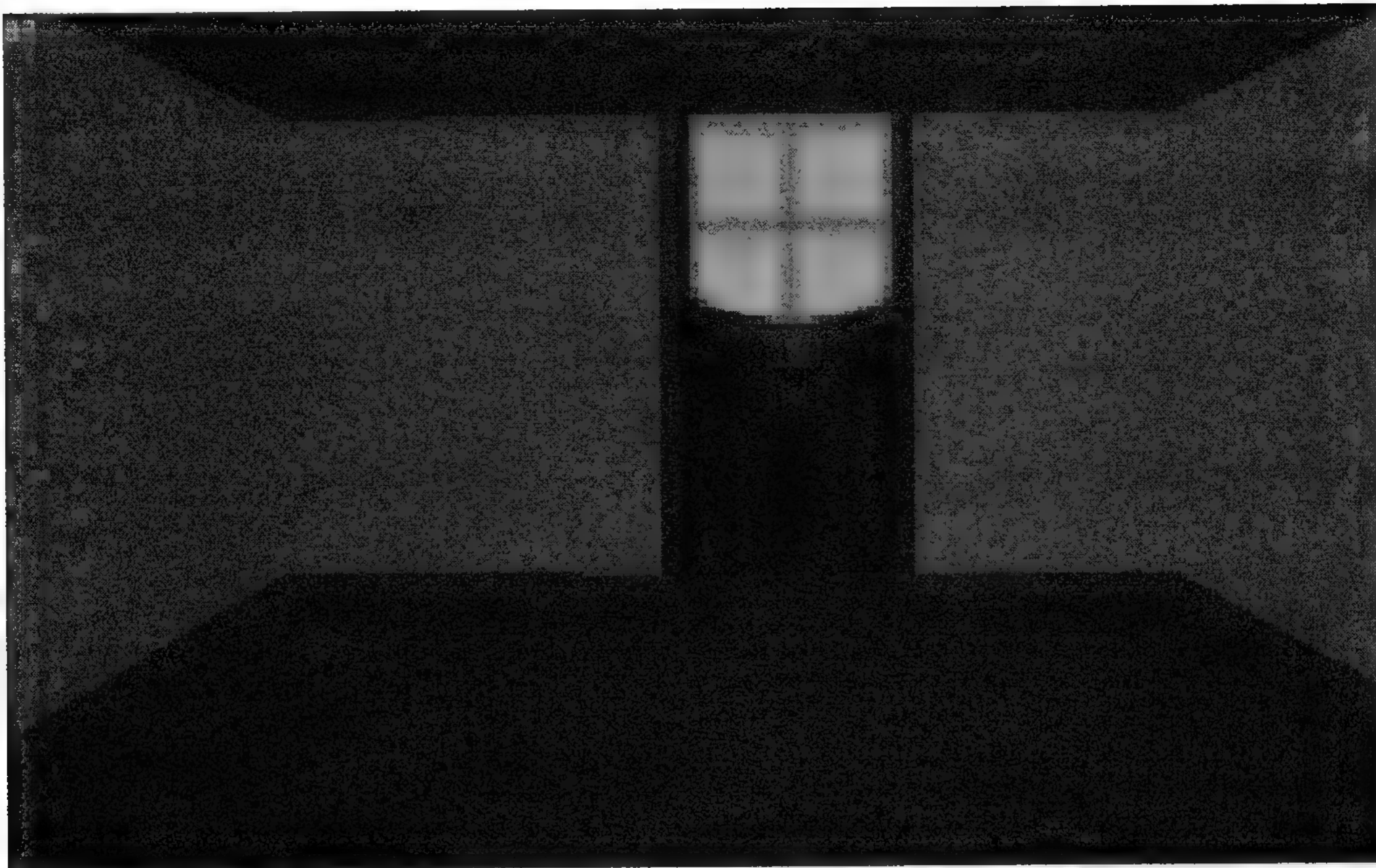


FIGURE 7.35: This wall light is made by lowering the ceiling of the small sector to the floor and applying the TECHLIGHT2 texture (#120).



NOTE

The small sector that you create for the purpose of holding the light texture can be any shape. The player will never see this sector because the ceiling is lowered to the floor.

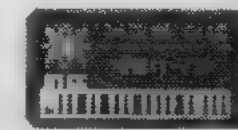
See Figure 7.35 for a visual representation of this area.

The next step is to give the wall that has the TECHLIGHT2 texture a unique LoTag value. This is very different from most of the other effects that you've worked with up to now because it requires that a *wall* be given a LoTag value. You give a wall a LoTag value the same way that you give a sprite a LoTag value: Place the mouse cursor on the wall while in 2D view mode, and press Alt + T to enter a LoTag

value (Alt + H to enter a HiTag value). You'll know you tagged a wall because the label will be red, whereas the label for sprites is either blue or purple, and the label for sectors is white.

Remember the unique LoTag value you gave the wall, because you'll need it for the next step, which is to add SE 3s to all the sectors you want affected by the lights being shot out. Give these SectorEffectors a LoTag value of 3 and the same unique HiTag value you gave the wall's LoTag value.

As with the ceiling light effect described previously, shade the walls of the sectors to the shade you want the sector to be after the lights are shot out, and shade the SectorEffector sprites to the shade you desire the sector to be *before* the lights are shot out. Your final room should resemble the one shown in Figure 7.35.

**NOTE**

Take note that in the wall light effect, a unique HiTag value is matched with a LoTag value, which is unusual. Usually two HiTags or two LoTags match to link objects.

RANDOM LIGHTS (SE 4)

Difficulty: Easy

You can make any sector flash in brightness and shade very easily. To do this, place a SectorEffector in the sector you want to flash at random, and give it a LoTag value of 4. Figure 7.36 shows an example. Set the shade of the sector walls, floors, and ceiling to a dark shade and the shade of the SectorEffector 4 sprite to a bright shade. You can also give the sector and the SectorEffector 4 sprite different palettes, and the sector will flash between those two palettes as well. This is a good effect for disco or alien areas (or alien disco areas!).



FIGURE 7.36: For the random lights effect, the brightness will fluctuate between the shade of the SectorEffector and the shade of the floor of the sector.

SUBWAY ENGINE (SE 6)

Difficulty: Hard (parts)

Subways are one of the great effects that separate *Duke Nukem 3D* from other first-person shoot-'em-ups. The ability to actually get onto a moving vehicle and move to another place really makes the game environment more realistic. Subways are composed of two parts: the *engine* and the *other* cars. This section describes how to create a single-car subway.

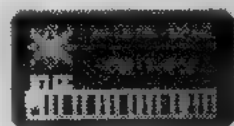
Start by making a huge sector to contain the subway. The entire subway must reside in one sector, so don't have visions of a huge subway traveling all over your giant cityscape. Also, subways won't work on slanted sectors, so keep the sector flat. (This shoots down all of those great *roller coaster* ideas I read about online as people started pushing the Build engine to its outer limits.)

Then, make a small sector on one side of the large sector and raise the floor a bit. Place a SectorEffector sprite in this sector. Give the SectorEffector a LoTag value of 6, and a HiTag value of 1. Also, angle this sprite so it points toward the front of the subway car. This sector will be the engine of the subway. You can put other sectors inside this sector and raise and angle their floors to shape the car any way you want. For this example keep the car a simple sector.

Now, you need to define the track the subway will follow. This is done with Locator sprites (6). Place Locator sprites in a path around the sector. After placing all the Locator sprites, give them all LoTag values in increasing numbers, starting with 0. That is, leave the first Locator sprite's LoTag value at 0, make the next LoTag value a 1, the next a 2, and so on, all the way to the last Locator sprite, as shown in Figure 7.37. You can also give a Locator sprite a HiTag value of 1; if you do so, the subway will stop at that Locator sprite for 5 seconds.

If you would like more than one set of cars on one track, give all the *sectors* that define the first set of cars a HiTag value of 1, the second set of cars a *sector* HiTag value of 2, and so on.

I labeled this effect a hard one to create for two reasons: (1) The Locator sprites must be placed in proper spots, or your subway car will spin around wildly or pass through the wall of the boundary sector or both. Try to remedy this by placing more Locator sprites around the curves of your imaginary subway track. (2) It is *vital* that the Locator sprites' HiTag values



NOTE

Subways move at a constant rate of speed, so the GPSpeed sprite has no effect on them.

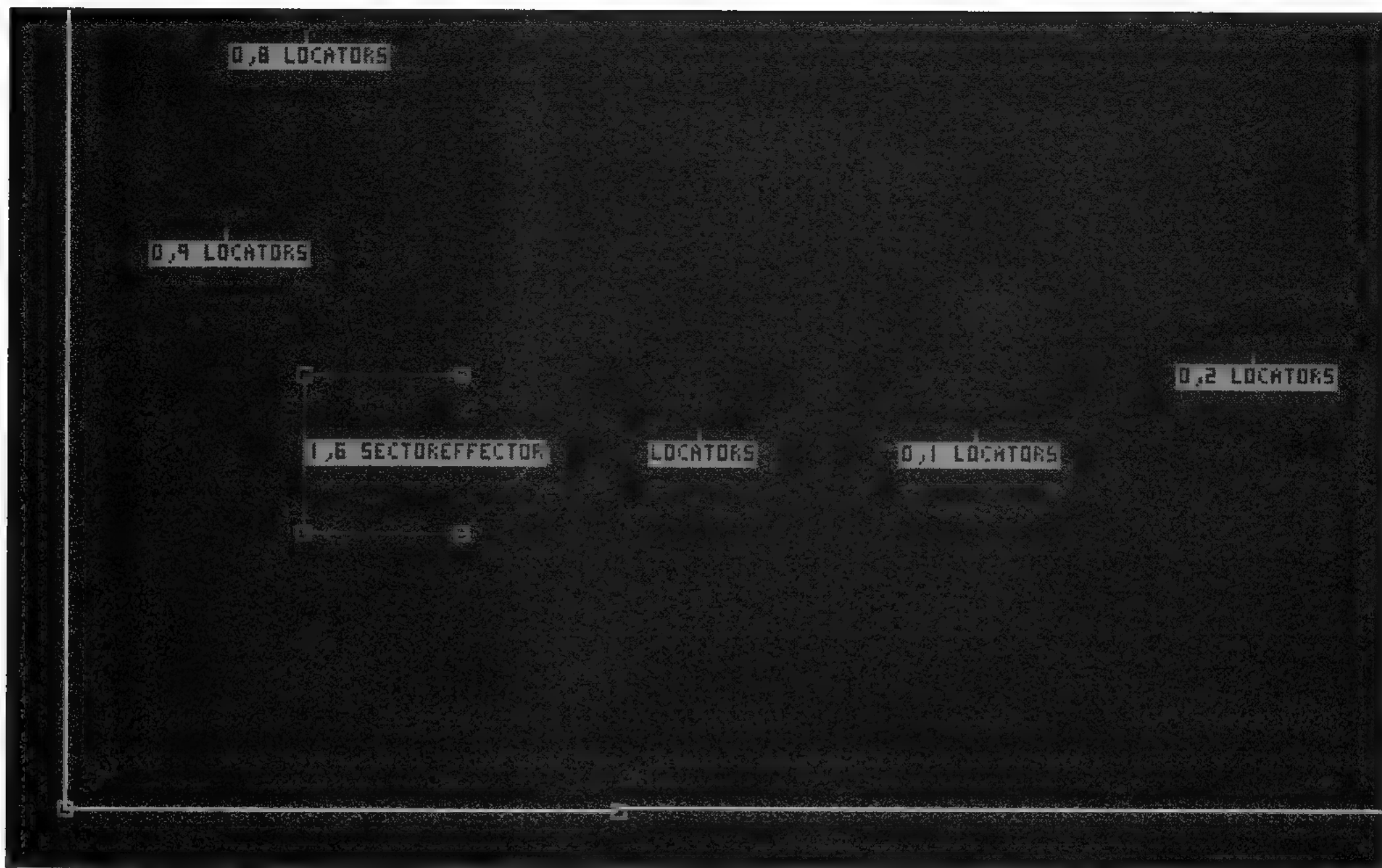


FIGURE 7.37: These subway sectors and Locator sprites define the track the subway will follow.

are numbered properly. Getting one out of order or skipping one will cause your subway car to move in very unexpected ways.

TELEPORT OR WATER (SE 7)

Difficulty: Easy to Medium

You've already seen an example of using an SE 7 sprite to set up two sectors that behave like an underwater sector and an above water sector. An SE 7 sprite can also be used to set up a standard teleporter between two places.

Start this effect by making a large room, and place two child sectors inside it, similar to the layout shown in Figure 7.38. The standard teleporter pad texture is #626, but there's absolutely no restriction on texture or even the size of the teleporter sectors. You can make *any sector* teleport the player.



SEE

See the section "The Above Water (ST 1) and Underwater (ST 2) Sectors" earlier in this chapter for a description of using the SE 7 sprite for a teleport effect between above water and underwater sectors.

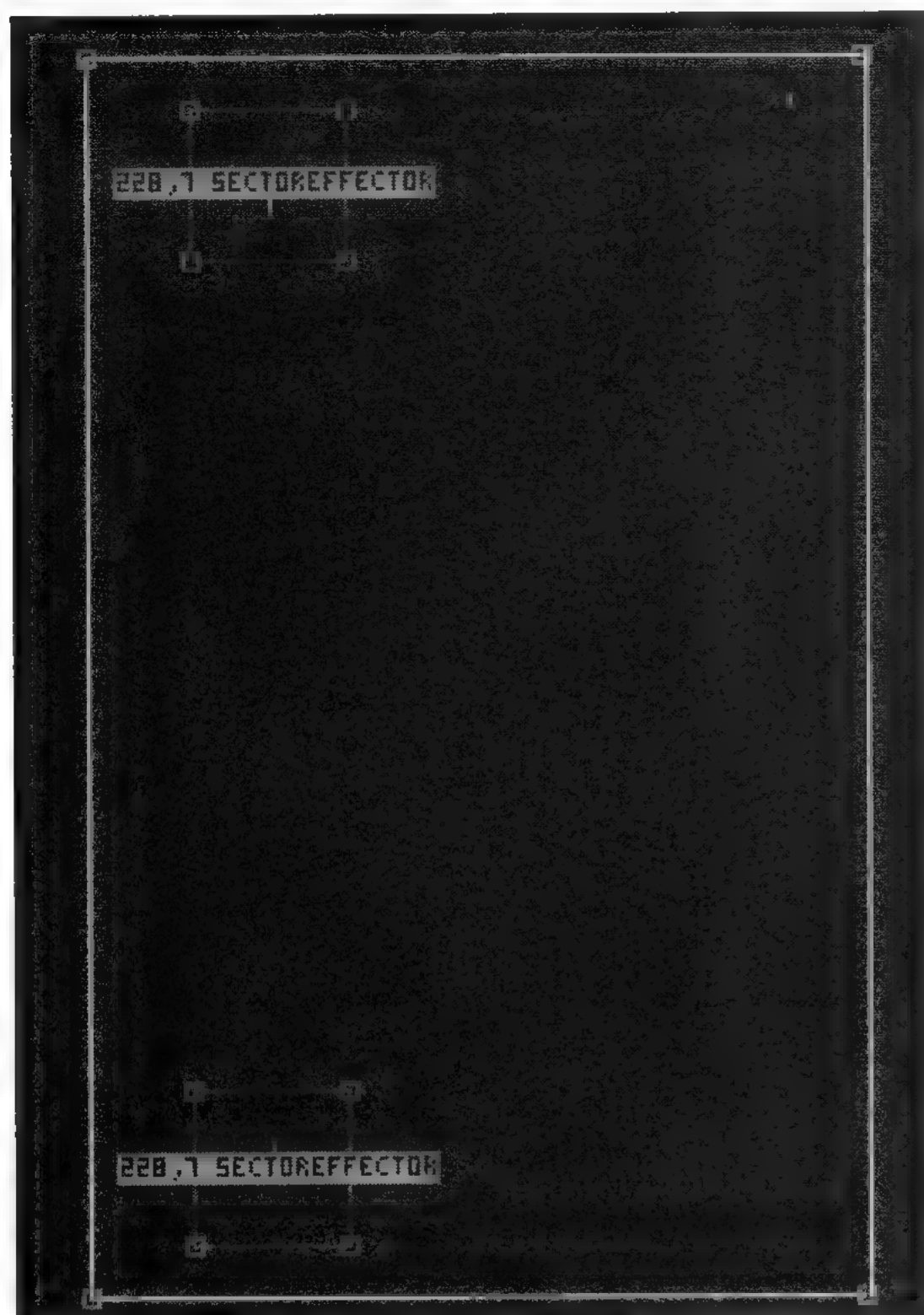


FIGURE 7.38: Although the two teleporter pads here are in the same room for clarity, they can be any distance apart on your level maps.

Next, place a SectorEffector sprite in each of the child sectors. Give each SectorEffector a LoTag value of 7, and a matching unique HiTag value that you haven't used anywhere else on the map. The angle of the SectorEffector specifies the direction that an object will be facing after it teleports.

Note that in the example shown in Figure 7.38, the angles of the SectorEffector sprites point toward each other. Two pads set up in this manner will cause a very cool effect. Try and duplicate this room exactly, and then go try it out in the game. Give yourself all the weapons, and then fire an RPG round into either teleporter. If you set up the pads correctly, the RPG round will travel between the two teleporter pads and never stop! (That is, until some unlucky sap stands in its path.) Note that you have to set the pads fairly far apart for this effect to take place, as shown in Figure 7.39. If the pads are too close together, they won't have time to reset, and the RPG round will fly straight through one of them.

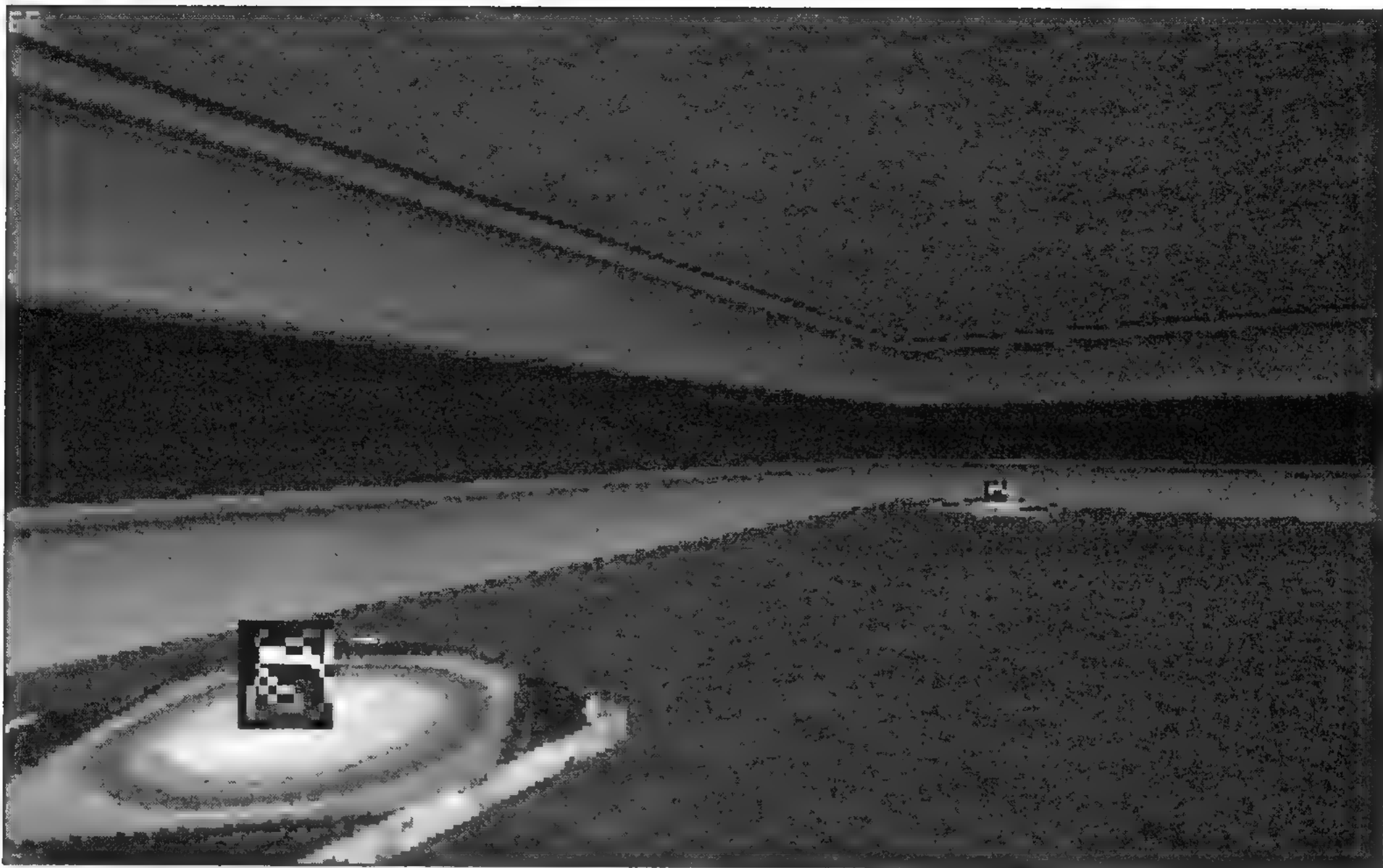


FIGURE 7.39: This 3D view of the teleport pads shows the second teleporter pad far off in the background.

There is one more transport effect that a pair of SE 7 sprites can perform. To demonstrate this example, take either one of the teleporter pads in the example room you've created and lower its floor about 40 clicks so that it's *well* below the level of the parent room (like a very deep pit). Then, take the SectorEffector that's on that pit sector and raise it several clicks, so that it's not on the floor anymore, but instead is suspended halfway down the shaft of the pit. Then, go to the other teleporter pad and raise the other SectorEffector so that it's nearly on the ceiling of the room. Now, go try out the level in the game. When you jump into the pit, you should fall for a bit, then land on the second teleporter pad! There won't be any teleporter noise or flash of light as before.

This silent teleport will happen any time the SE 7s are not on the floor of their sectors. It is the effect used at the very beginning of the game in the Hollywood Holocaust level (E1L1), where the player starts on a rooftop, jumps into an air shaft (where the fan was before it explodes), and lands on the street below. If you load that level into Build, you'll see that the roof sectors and the street sectors where the player lands are *not* physically over one another—they are in different places on the map. The SE 7s handle the silent teleport between the two areas.

SECTOR LIGHTS WHEN DOOR OPENS (SE 8)

Difficulty: Easy to Medium

This effect creates a brighter room that sheds light into a darker room when a door between them opens. It is an effect you should use wherever appropriate because it really enhances the realism of the environment.

This effect can be used with any door that opens by raising a ceiling—the most common being SE 20, the ceiling-based door. Start by creating two room sectors separated by a ceiling-based door. Make one of the rooms dark by shading the walls, floor, and ceiling. Then, place two SectorEffector sprites: one in the door sector and one in the dark sector. Give each a LoTag value of 8, and the same unique HiTag value. The shade of the SectorEffector in the dark sector will be the shade that the light becomes when the door opens. An example is shown in Figure 7.40.

You can have multiple sectors affected by the same door. To do so, just create each sector with its own SectorEffector that has a LoTag value of 8 and the matching HiTag values. If you look at the example of the SectorEffector in the file `_SE.MAP`, you'll see that the designer created bands of thin sectors with shades that decrease in value as the

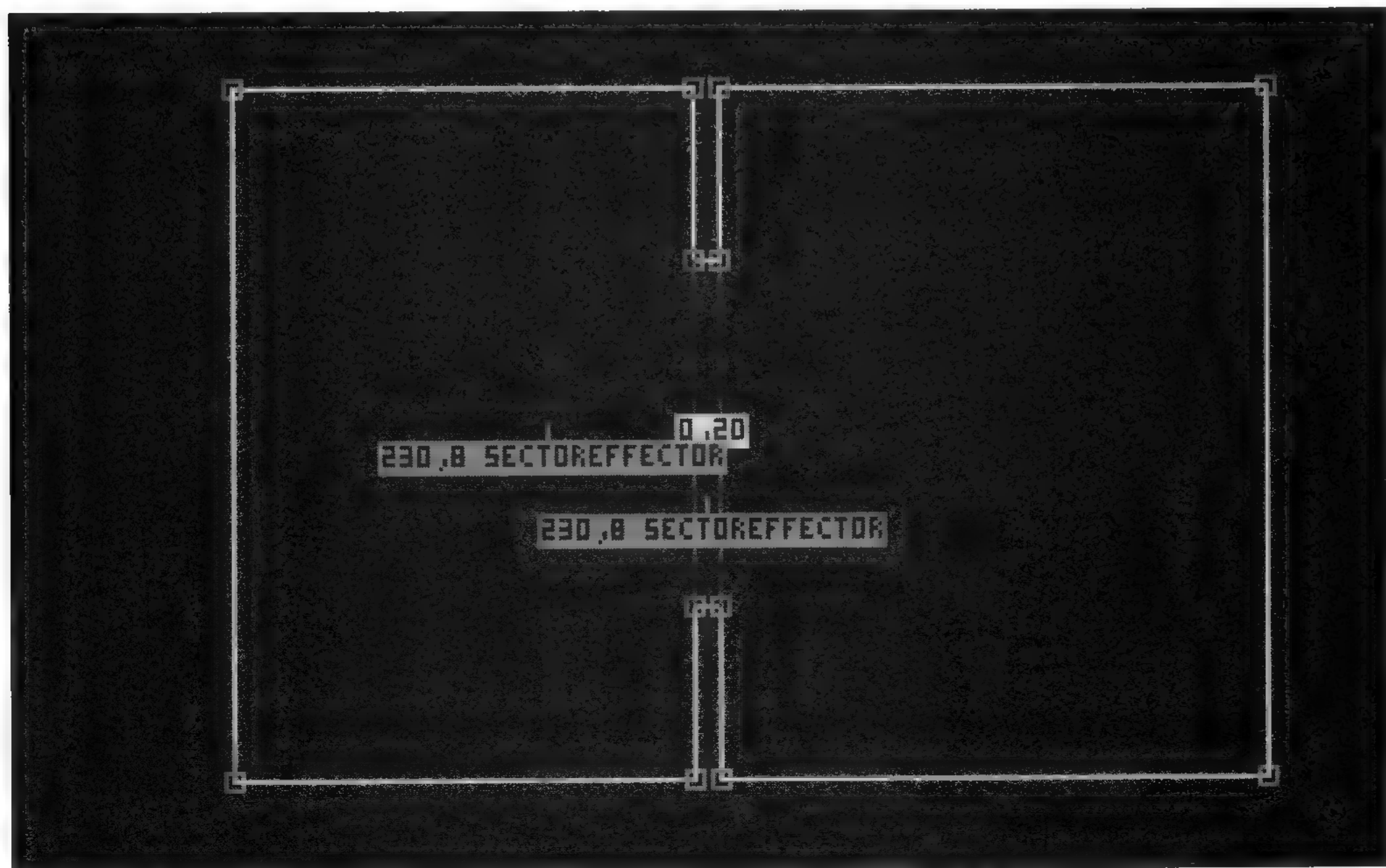


FIGURE 7.40: Here's how to set up a dark room (left) that will brighten when the door opens.

sectors get farther from the door. This makes it look like the light streaming through the door gets dimmer and dimmer.

SECTOR LIGHTS WHEN DOOR CLOSES (SE 9)

Difficulty: Medium

This effect is similar to the effect created by the SE 8 sprite, but the lights appear to be on the wall of the door itself. This causes the lights to brighten in the sector when the door closes.

To create this effect, make a ceiling-based door with ST 20. However, leave the door *open*. Then, put an SE 9 sprite in the door sector and in the bordering dark room's sector. Give both SE 9s a unique HiTag value. The room will light up when the door closes, as if the lights were on the door itself. Put a light texture (like #120) on the door to make it appear as if a bank of lights are mounted on the face of the lowering door.

In 2D view mode, this effect looks very similar to the one you saw in Figure 7.40. The only difference is that the SectorEffectors have LoTag values of 9 instead of 8. As you can see in Figure 7.41, the door is shown half open in 3D view mode with the TECHLIGHT2 texture applied.

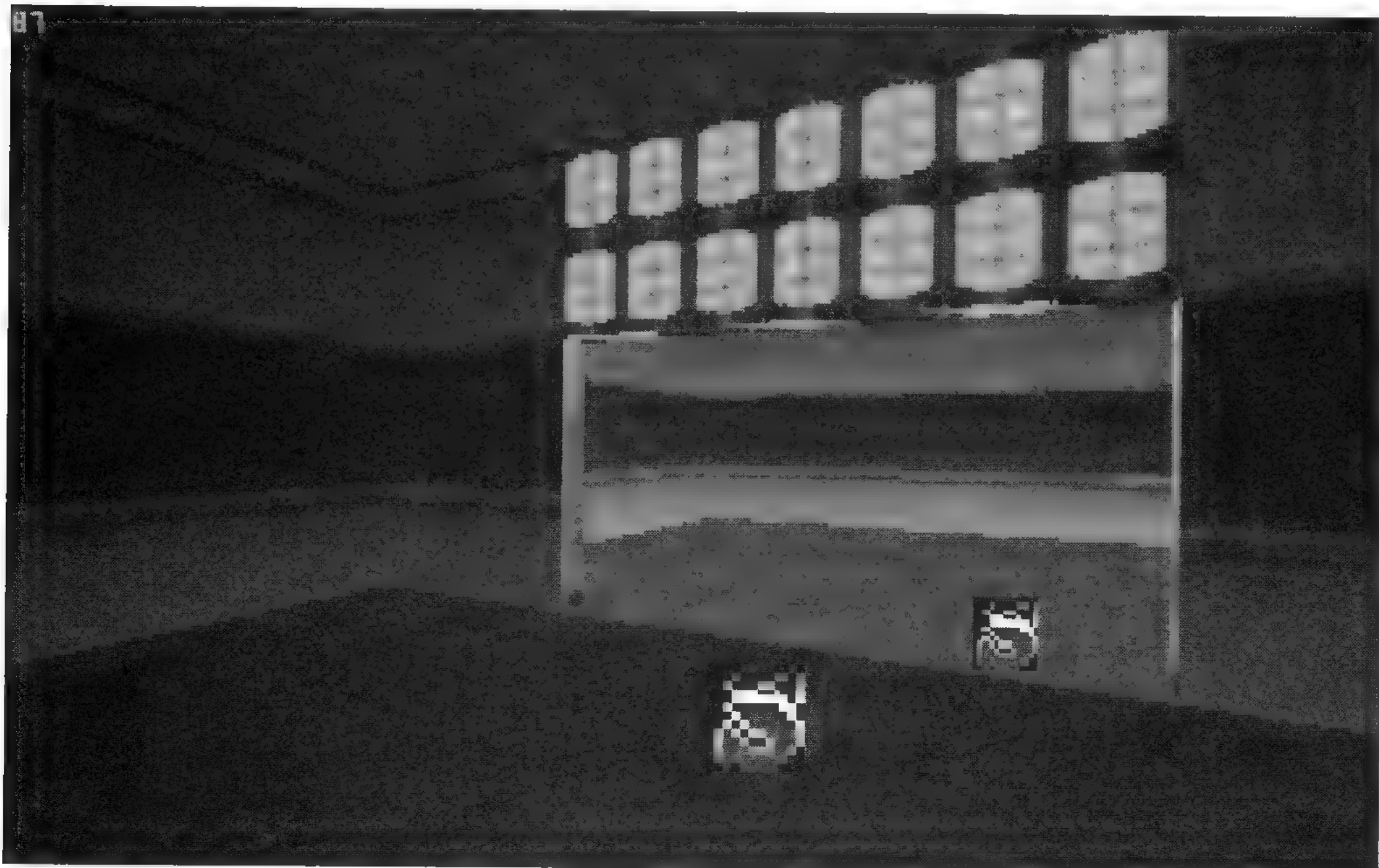


FIGURE 7.41: Here is the door effect created by SE 9 sprites shown in 3D view mode.



DOOR AUTO-CLOSE (SE 10)

Difficulty: Easy

The SE 10 sprite is used with any of the door Sector Tags to make the door close automatically after a set length of time. Refer to any of the previous sections covering any of the door Sector Tags for a description of using the SE 10 sprite.

SWINGING DOOR PIVOT POINT (SE 11)

Difficulty: Medium to Difficult

The SE 11 sprite is used as a pivot point for a swinging door, which is ST 23. See the section “The Swinging Door (ST 23),” earlier in this chapter, for a description of using SE 11 for a swinging door’s pivot point.

LIGHT SWITCH (SE 12)

Difficulty: Easy

To make a switch that turns on the lights, first design the sector or sectors in which you want the lights to turn on. Shade the walls and floors of these sectors so that they’re dark. Now, add a Switch sprite (sprite #712 is the standard light switch) to a wall somewhere. The switch doesn’t have to be anywhere near the sectors it lights up. Give the Switch sprite a unique LoTag value (x). Then, add a SectorEffector to each sector that will light up. Make the SectorEffector’s HiTag value x and its LoTag value 12, as shown in Figure 7.42. Also, the shade of the SectorEffector will be the shade of the sector when the lights go on, so you could have one light switch brighten several sectors, and each sector can brighten to a different value.

C-9 EXPLOSIVE (SE 13)

Difficulty: Hard

Holes in walls caused by explosions are another special effect where you design the look of the area after the explosion, and *Duke Nukem 3D* fills in the sector automatically to look flat before the explosion happens.

To create an exploding hole, begin by creating a room to stand in and watch the hole. Then, create three small sectors that connect to the room and to each other, in a layout similar to that shown in Figure 7.43. Angle the floors and ceilings of these three sectors so that they look like a hole. (For example, angle the two outer sectors’ ceilings

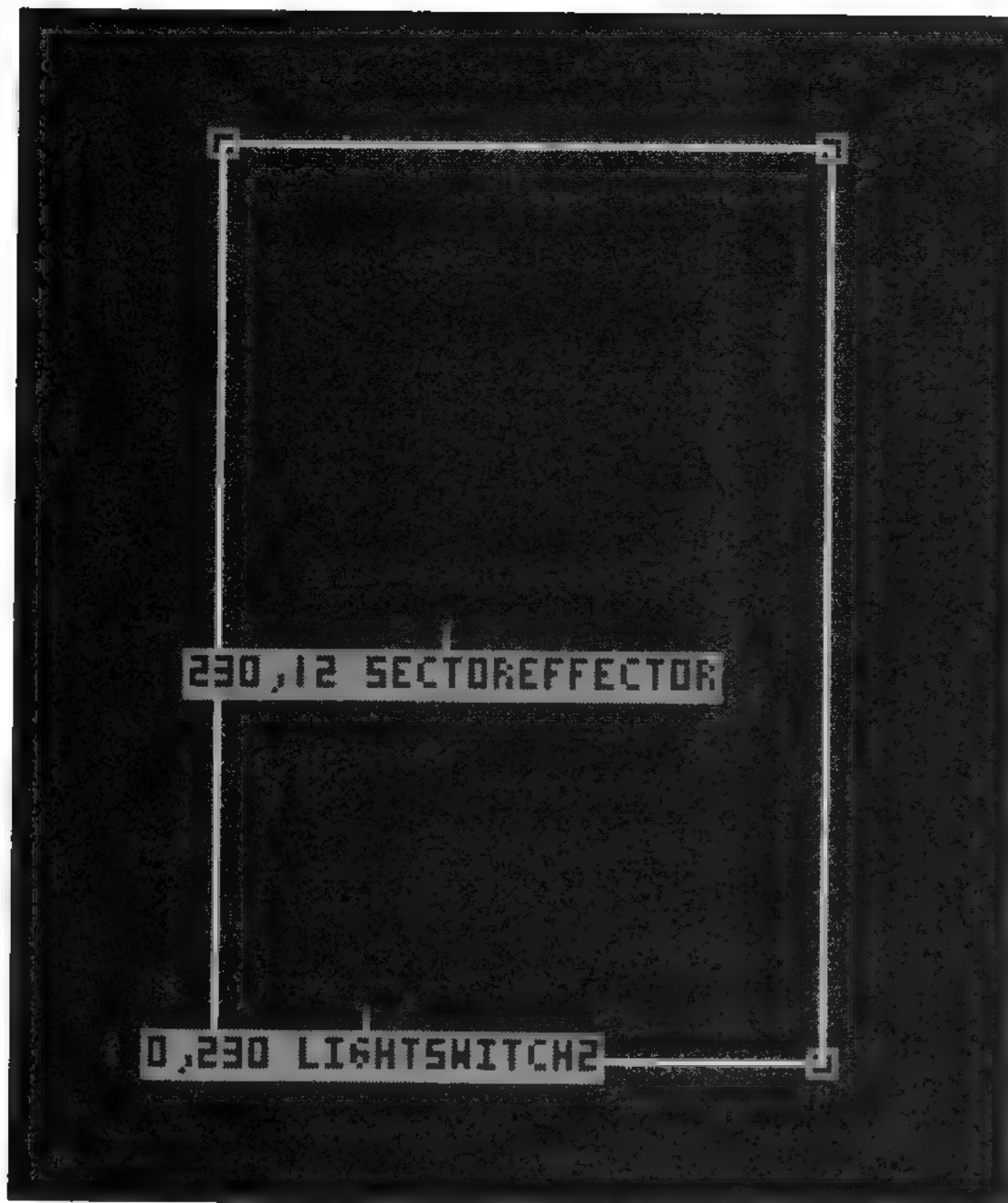


FIGURE 7.42: In this sector layout for a simple light switch, the LoTag value of the switch matches the HiTag value of the SE 12 sprite.

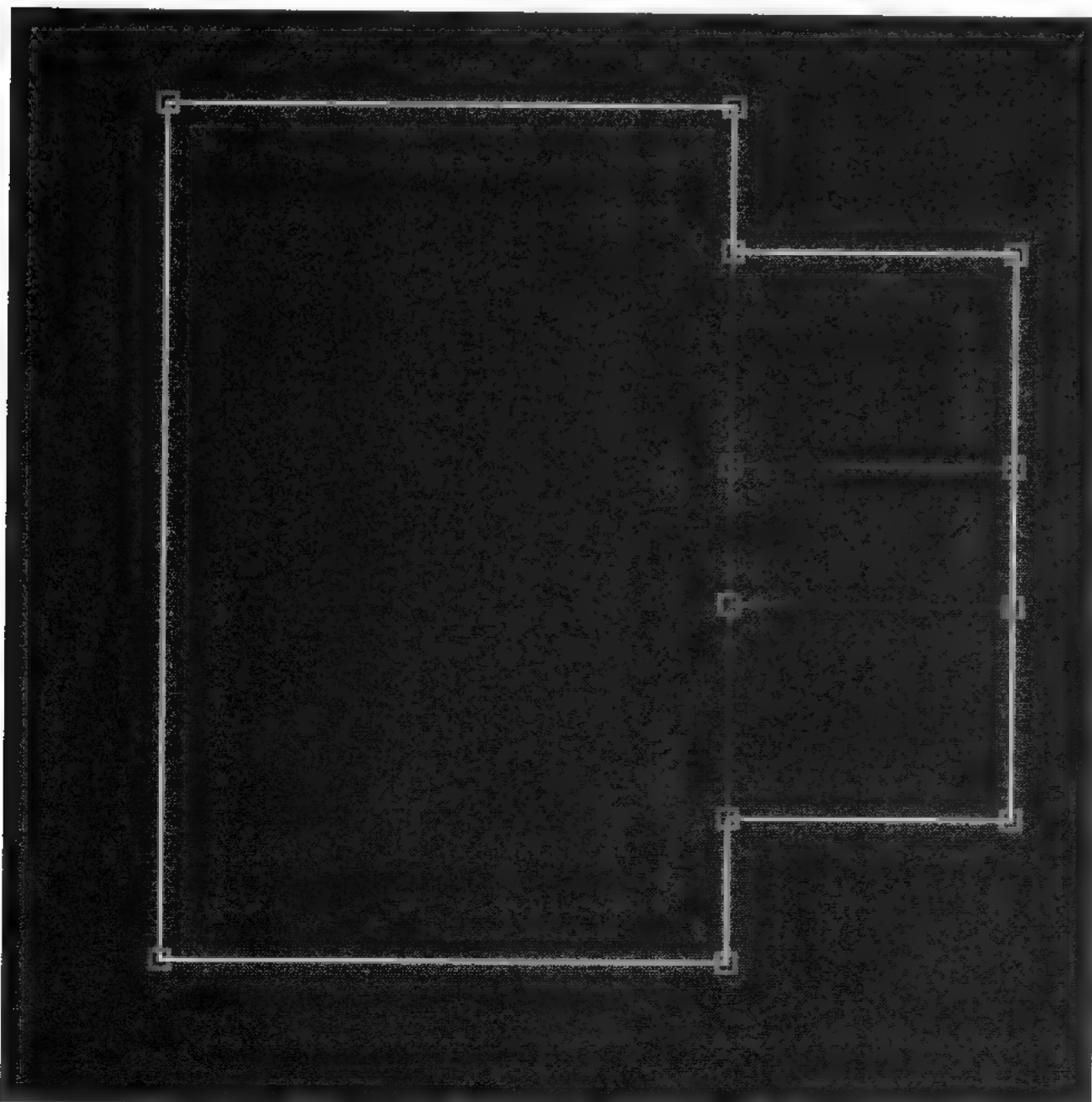


FIGURE 7.43: This is a sector layout for creating the exploding wall effect.

and floors inward toward the middle sector.) You can also find a good rough texture to look like blasted rock (texture #240 is good for this).

Next, put a Crack sprite somewhere (sprites 546–549 are cracks) near the hole. Make sure it lies flat against a wall, even if it's one of the invisible two-sided walls that join the room to one of the hole sectors. Don't worry if the crack seems to float in space if you put it right over the hole. *Duke Nukem 3D* will put a texture there automatically when the game starts. Give the Crack sprite a unique HiTag value. Next, place a SectorEffector sprite in each of the three hole sectors. Give each a LoTag value of 13 (for the C-9 Explosive), and the same HiTag value you assigned the Crack sprite.

Place an actual C-9 sprite (sprite 1247, called SEENINE in DEFS.CON) in each of the three hole sectors. The LoTag value of each C-9 Explosive sprite will be the time lag before it explodes. You can make each LoTag value different to stagger the explosions. Also, give each C-9 Explosive sprite the same HiTag value you assigned to the Crack sprite. Figure 7.44 shows you what the final effect should look like in 2D view mode.

When you try out your exploding wall effect in the game, you might notice that when the hole first appears, you can see the C-9 canisters before they explode. To eliminate this, simply scale each C-9 Explosive sprite by decreasing its width until you can't see it anymore.

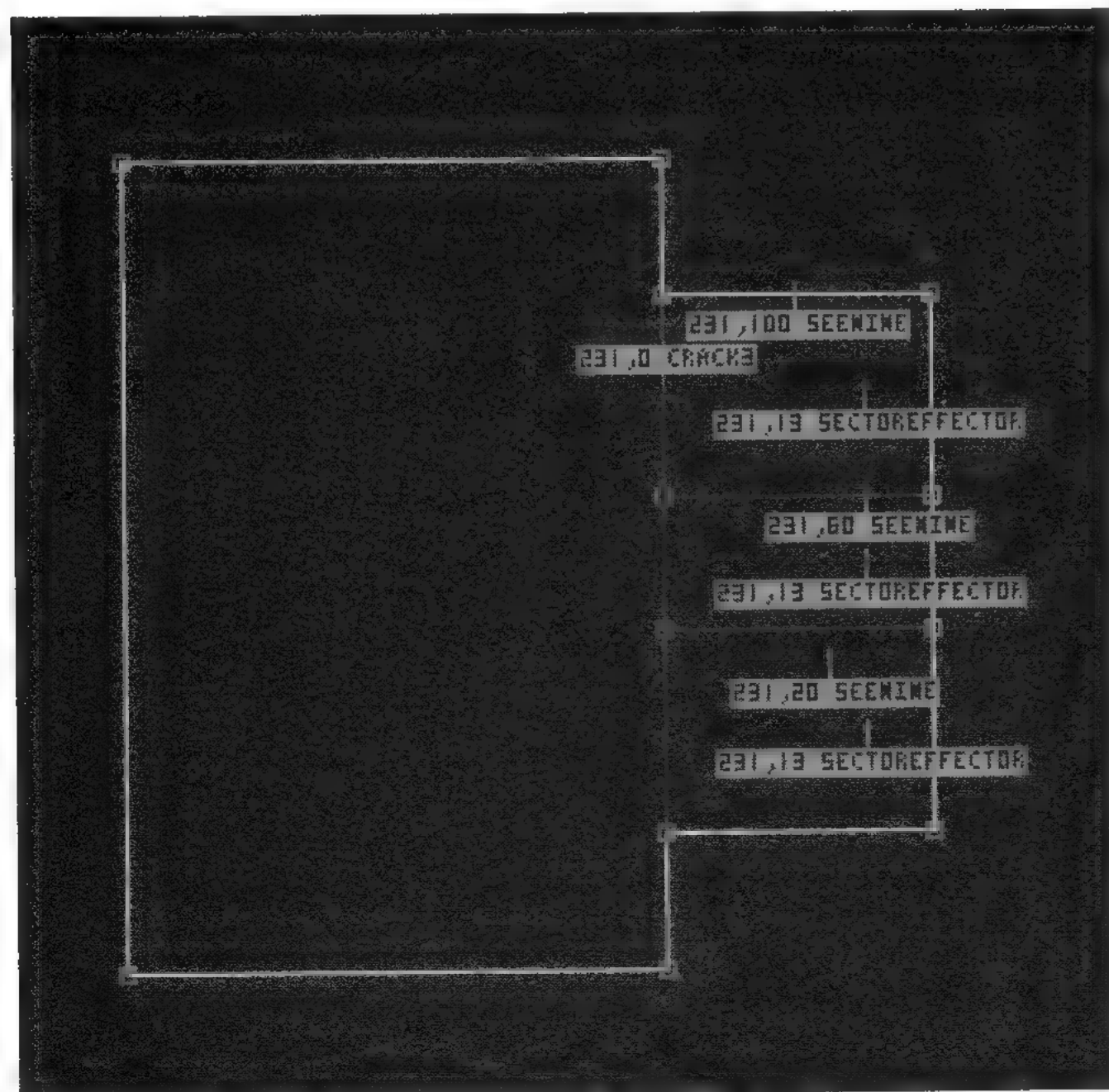


FIGURE 7.44: In the completed sectors for the exploding wall effect, the C-9 Explosive (SEENINE) sprites have different LoTag values, so that each explodes at a different time.

One additional problem you might run into when you try out the area in the game is that one of the hole sectors won't be completely covered up by the Build engine. Instead, you will be able to see part of the hole before triggering the explosion. If this is the case, you need to adjust the first wall of that sector until it is the lowest of all the walls that make up the sector. The example here has the sector's first wall higher than the other walls (due to the slope in the sector). Simply flip the sector's first wall to the opposite side, then readjust the slope to make the hole again.

Another Trick for Making Holes in Walls

If the hole you plan on making is made up of a few dozen small sectors, and you dread the thought of placing SectorEffectors and C-9 Explosive sprites into each of those small sectors, there's a shortcut you can use to disguise your hole by using a single sector. The trick is to place a very thin sector directly against the wall that has the hole. Place the SectorEffector and C-9 Explosive sprite in this one sector. Build will automatically bring the ceiling of this sector down to the floor, obscuring the hole. When the hole is triggered, the wall will disappear, revealing the hole.

SUBWAY CAR (SE 14)

Difficulty: Medium to Hard

Making a single subway car was already discussed for SE 6. If you want a multiple-car subway, simply make more cars behind the main one. (Consider copying the first car to make them.) Then, change the SectorEffector in the cars to type 14 instead of 6, that is, change the LoTag values from 6 to 14 on all the cars except the first one (see Figure 7.45). Make sure the HiTag values for the subway cars' SectorEffectors match the HiTag value of the subway engine's SectorEffector and that the angles of all the SectorEffectors point in the same direction.

THE SLIDE DOOR (SE 15)

Difficulty: Medium

This sprite is used exclusively with ST 25, the slide door effect. See the section "The Slide Door (ST 25)," earlier in this chapter, for a discussion of using this SectorEffector with ST 25.

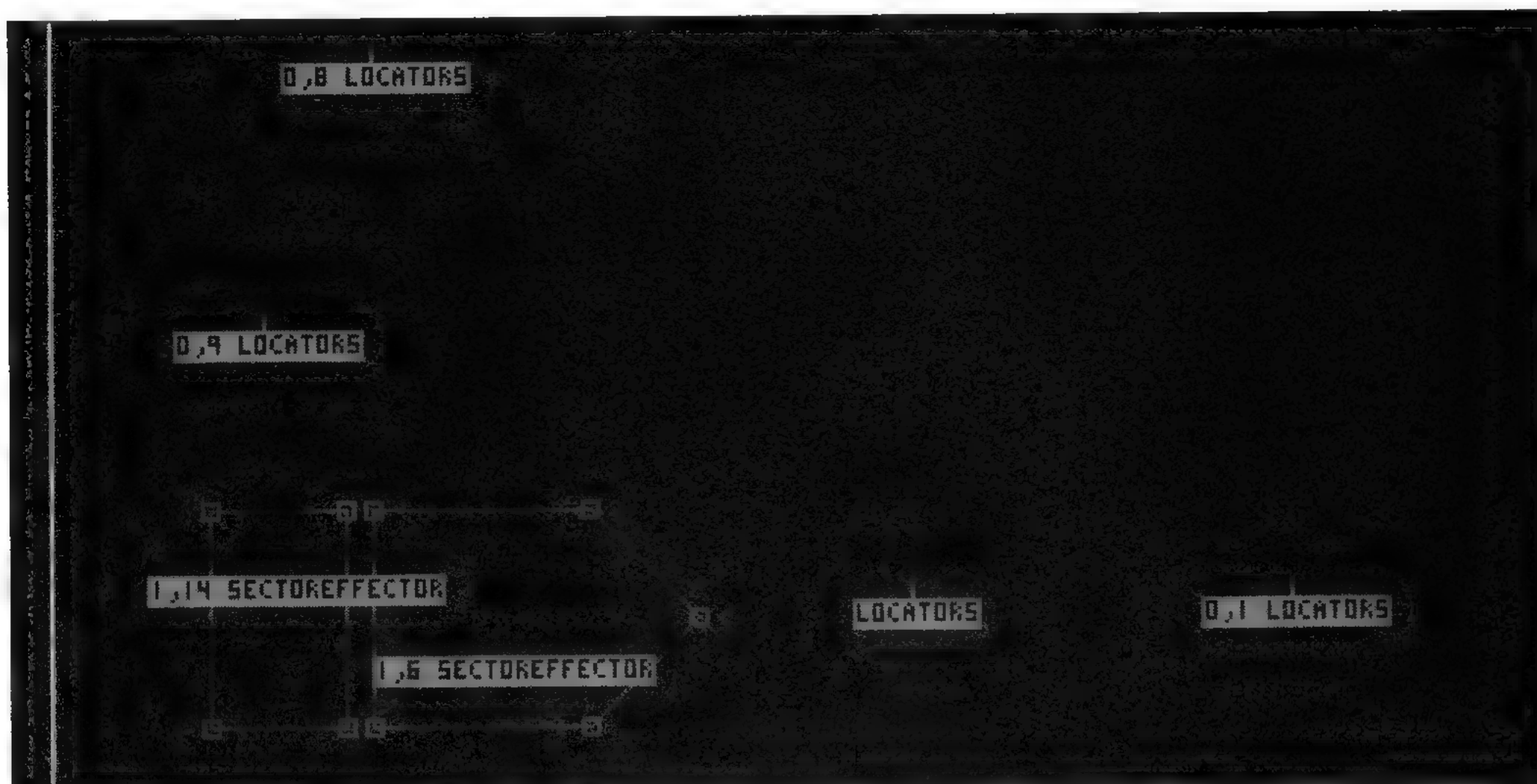


FIGURE 7.45: This effect contains a subway train with one engine and one car.

THE ELEVATOR TRANSPORT (SE 17)

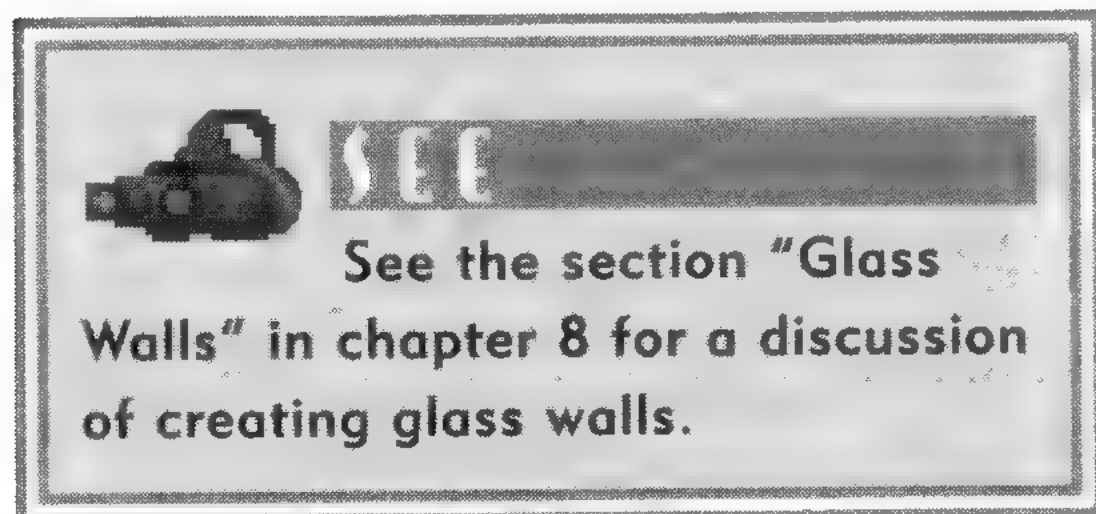
Difficulty: Hard

This sprite is used exclusively with ST 15, the elevator transport effect. See the section “Elevator Transport (ST 15),” earlier in this chapter, for a discussion of using this SectorEffector with ST 15.

SHOT TOUCHPLATE CEILING DOWN (SE 19)

Difficulty: Medium

SE 19, named “Shot TouchPlate Ceiling Down,” will cause a ceiling to drop if a bullet hits a sector. To create this effect, create a glass wall effect. Then, add a SectorEffector sprite to the sector the glass wall is in, and assign a LoTag value of 19 to it. When you shoot and destroy the glass with a rocket or other explosion, the ceiling of the sector will lower.



THE BRIDGE (SE 20)

Difficulty: Medium

This sprite is used exclusively with ST 27, the bridge effect. See the section “The Bridge (ST 27),” earlier in this chapter, for a discussion of using this SectorEffector with ST 27 to create a bridge effect.

THE DROP FLOOR (SE 21)

Difficulty: Medium

This sprite is used exclusively with ST 28, the drop floor effect. See the section “Drop Floor (ST 28),” earlier in this chapter, for a discussion of using this SectorEffector with ST 28 to create a dropping floor or ceiling effect.

THE TEETH DOOR (SE 22)

Difficulty: Medium to Hard

This sprite is used exclusively with ST 29, the teeth door effect. See the section “Teeth Door (ST 29),” earlier in this chapter, for a discussion of using this SectorEffector with ST 29 to create a teeth door effect.

CONVEYOR BELT OR WATER CURRENT (SE 24)

Difficulty: Easy

Conveyor belts are sectors with floors that move in a certain direction, which in turn move any objects standing on them. They are very easy to make. First, make any size sector. Put a long child sector inside it, as you see in Figure 7.46. Raise the floor of the child sector a bit. This will be the conveyor belt.

Place a SectorEffector sprite inside the child sector. Give the SectorEffector a LoTag value of 24, and change the angle of the SectorEffector to point in the direction you want the conveyor belt to travel. You can also add a GPSSpeed sprite to speed up or slow down the speed of the belt. The LoTag value of the GPSSpeed sprite controls the speed of the conveyor belt.

You’re done! Try out the conveyor belt in the game. You can also use the same SectorEffector sprite in an above water sector to make the water have a current that travels in a certain direction. Simply add the SE 24 sprite and the GPSSpeed sprite to an existing above water sector, and that watery section will have a current.

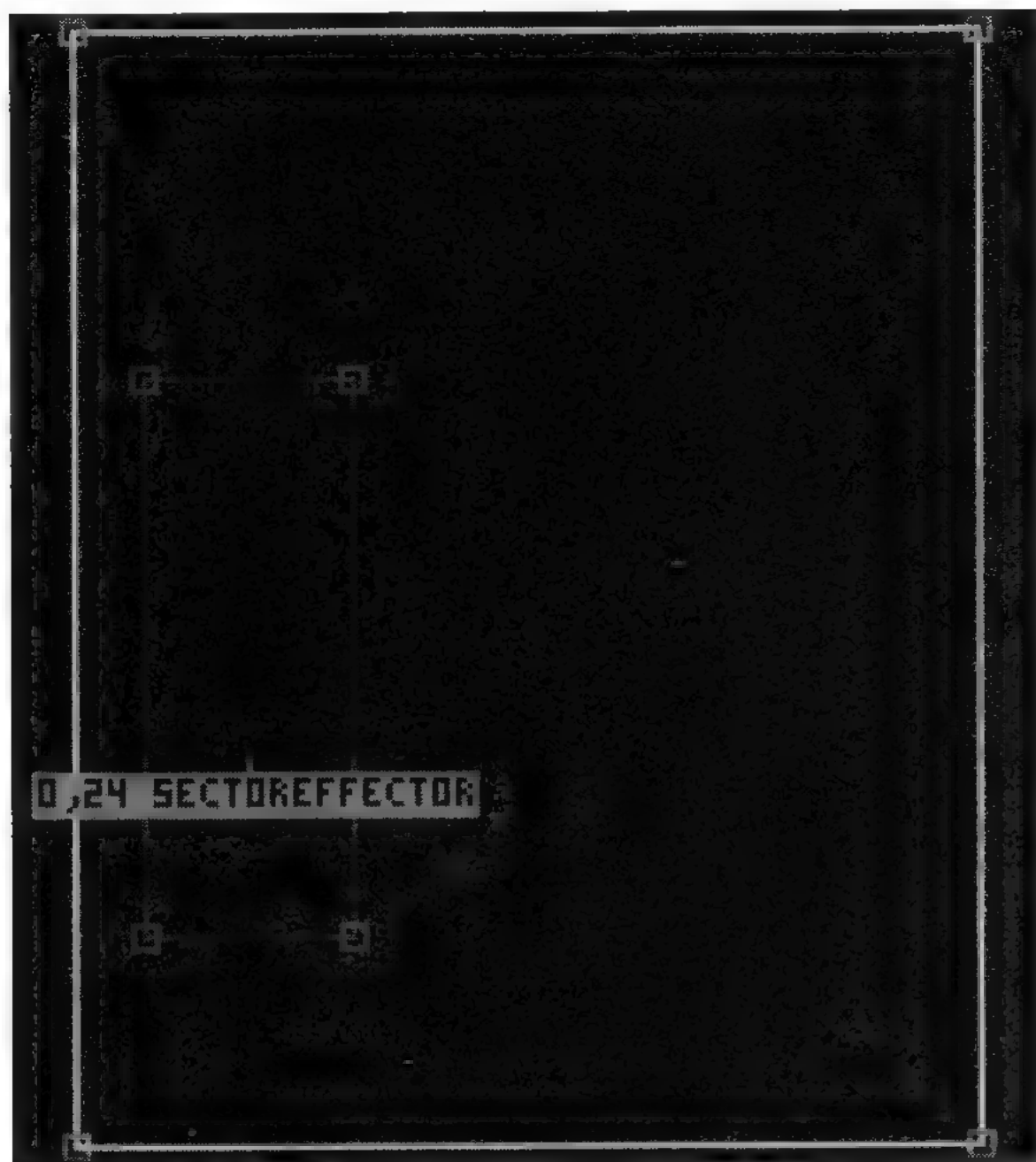


FIGURE 7.46: Here's the sector layout for creating a conveyor belt effect.

ENGINE PISTON (SE 25)

Difficulty: Easy

An engine piston sector is really just a sector with a ceiling that moves up and down quickly. The ceiling will also crush a player or monster standing under it. Start this effect by making any size room and placing a child sector inside it, as you see in Figure 7.47. The child sector will be the piston. Then put a SectorEffector sprite inside the child sector, and give it a LoTag value of 25. Now, switch to 3D view mode, and make sure the height of the SectorEffector is different from the parent sector's ceiling height. The piston will move between the current ceiling height and the height of the SectorEffector.

You can also put a GPSPeed sprite in the piston sector to control its speed or add a Music&SFX sprite to have the engine piston make a sound.

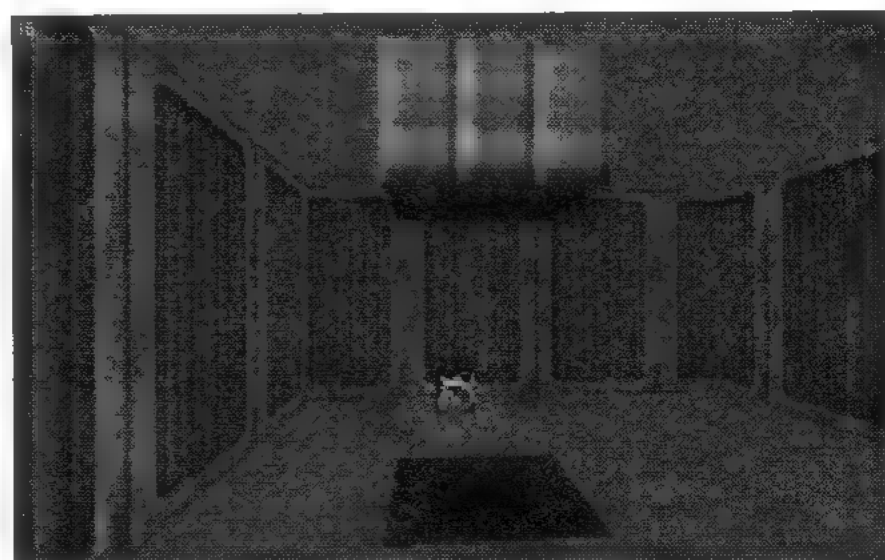




FIGURE 7.47: Here are the sectors for the engine piston effect.

Although the description of this SectorEffector describes it as an engine piston, it can really be used on any sector where you'd like the ceiling to move up and down to crush the player. Using your imagination here should result in all kinds of devious traps, especially if you include Activator and TouchPlate sprites.

CAMERA FOR DEMO RECORDING (SE 27)

Difficulty: Easy

When you see the demos at the beginning of the game, sometimes the view of the demo is a first-person view, sometimes it is from a “chase camera” that seems to follow the character around, and sometimes it is from some other outside source, as if from a security camera mounted on a wall. You can set up these demo-recording cameras wherever you want them in your level, and the demo's view will automatically switch to the outside view whenever the player gets within the camera's radius.

To set up such a camera, place a SectorEffector wherever you'd like the demo camera. Give the SectorEffector a LoTag value of 27 and a HiTag value that represents the range the camera will have. One neat trick is to place your SE 27 sprites near the other types of security cameras (the ones that connect to the monitors), and it will appear like those cameras are the ones recording the demos.

FLOATING SECTOR OR WAVES (SE 29)

Difficulty: Medium to Hard

You can create an effect with waves using SectorEffector 29. The hardest part about waves for me was figuring out how to make connecting child sectors. After 30 minutes or so of banging my head against the monitor, I finally figured it out. In addition to the discussion here, you can check out the wave sectors that are provided in the file `_SE.MAP`.

To begin this effect, make an outer sector to house the wave sectors. Then, create a rather large child sector within it. To make this child sector into several small ones, split each of the longer walls (insert vertices) with the `Ins` key. Then, press the spacebar on one of the new vertices, and draw to the other vertex on the opposite wall. The sector will be split. Do this a few more times until you have three or four child sectors lined up next to each other, as shown in Figure 7.48.

Then, lower each of the connecting child sectors' floors a bit, and apply to each the floor texture #336, so that the area looks like a pool of water. The first walls of the wave sectors will *anchor* each sector to the sector with which it connects. You should make every first wall on the same side. For example, make each sector's first wall the

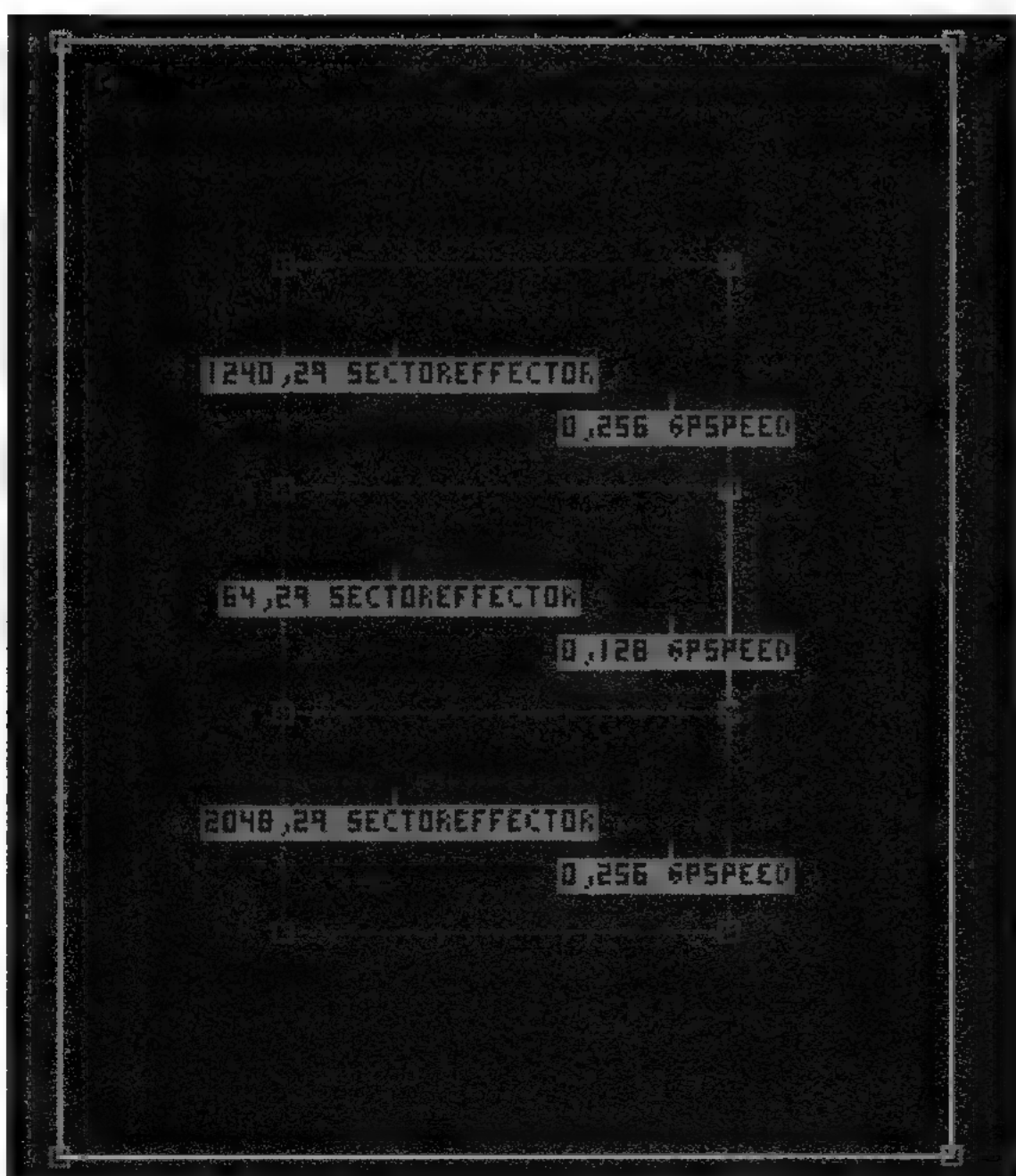
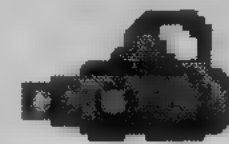


FIGURE 7.48: To create a realistic wave effect, you will usually need to line up several wave (child) sectors near each other.

northern wall of each sector, as shown in Figure 7.48.

Now, each wave sector needs two sprites placed inside it to make the waves work. The first is a SectorEffector 29 sprite. Add a SectorEffector, make the LoTag value 29, and make the HiTag value represent the *shape* of the wave. Valid values for the wave shape are 0 through 2048. You will need to experiment to determine which wave shape you like best. The second sprite each sector needs is a GPSPeet sprite. The LoTag value for these sprites will represent the height of the wave, so you can vary the values between each wave sector to create waves of different sizes.

Your wave sectors can become above water sectors as well (ST 1), and they can be connected to an identical set of underwater sectors.



SEE

See the section "The Above Water (ST 1) and Underwater (ST 2) Sectors," earlier in this chapter, for creating an above water and underwater sector pair.

TWO-WAY TRAIN (SE 30)

Difficulty: Medium to Hard

This sprite is used exclusively with ST 31 for the two-way train effect. See the section "Two-Way Train (ST 31)," earlier in this chapter, for a discussion of using this SectorEffector with ST 31 for a two-way train effect.

FLOOR MOVE (SE 31)

Difficulty: Medium to Hard

This SectorEffector is used to raise or lower a floor when activated by a Master-Switch or Activator sprite. The floor could be used as another type of lift, for example, or it could move to reveal a secret area behind the moving sector. Start this effect by creating a room and a child sector inside it, as shown in Figure 7.49. Raise the floor of the child sector a few clicks so you can easily differentiate it from the rest of the room. Then, put a SectorEffector sprite in the child sector. Give the SectorEffector a LoTag value of 31.

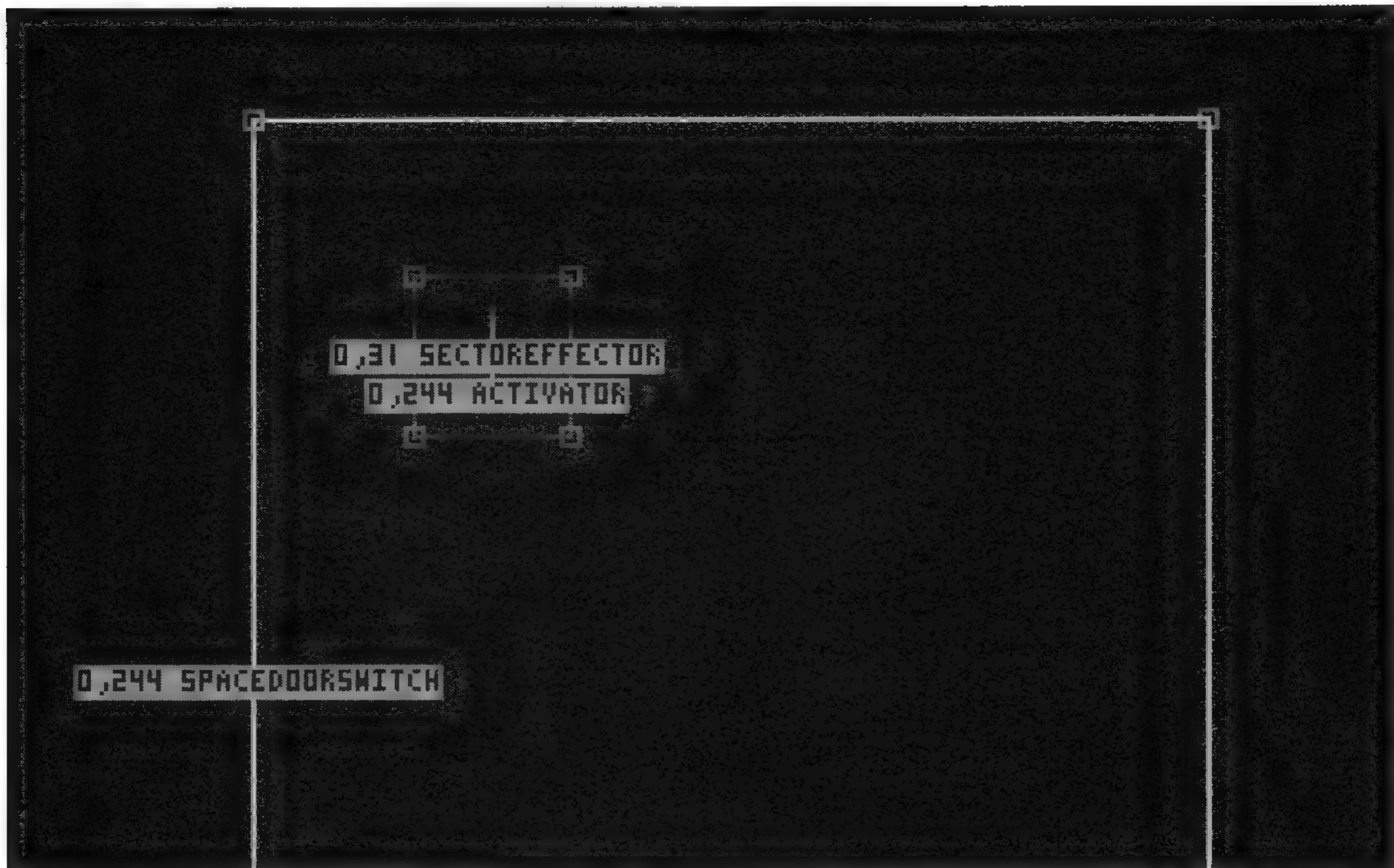


FIGURE 7.49: The sector layout for creating a moving floor using SE 31 and Switch sprites.

Next, switch to 3D view mode, and change the height of the SectorEffector so that it's different from the current floor height. Return to 2D view mode, and decide how you want the floor to move:

- ❖ Make the floor move *from* its current height *to* the height of SE 31, with the SectorEffector's angle pointing downward (south), or
- ❖ Make the floor move *from* the height of SE 31 *to* the designed height, with the SectorEffector's angle pointing upward (north).

These choices allow you to set up the floor and the SectorEffector so that you never have to put a SectorEffector sprite below the level of the floor, making it impossible to see in 3D mode.

The next step is to place an Activator sprite in the sector, and to give the Activator sprite a unique (unused) LoTag value (x). Finally, place a Switch sprite on a wall somewhere nearby, and give it the same LoTag value that you assigned to the Activator sprite's LoTag value (x). Your floor will now move as you designed it when you flip

the switch. It will move back to its original position when the switch is flipped again. Figure 7.50 shows a 3D view of this effect.

CEILING MOVE (SE 32)

Difficulty: Medium to Hard

This SectorEffector is used to raise or lower a ceiling when activated by a MasterSwitch or Activator sprite. This effect works very similarly to the floor move effect, except for the use of the angle of the SectorEffector sprite. Start this effect just as you would for a moving floor effect, by creating a room with a child sector inside it. Change the height of the ceiling of the child sector so you can easily see it. Then, put a SectorEffector sprite in the child sector, and give it a LoTag value of 32.



NOTE

Although it is possible to lower a sprite to below the level of the floor, you should avoid doing so whenever possible because you have to lower the floor of the sector in 3D mode to see the sprite again.

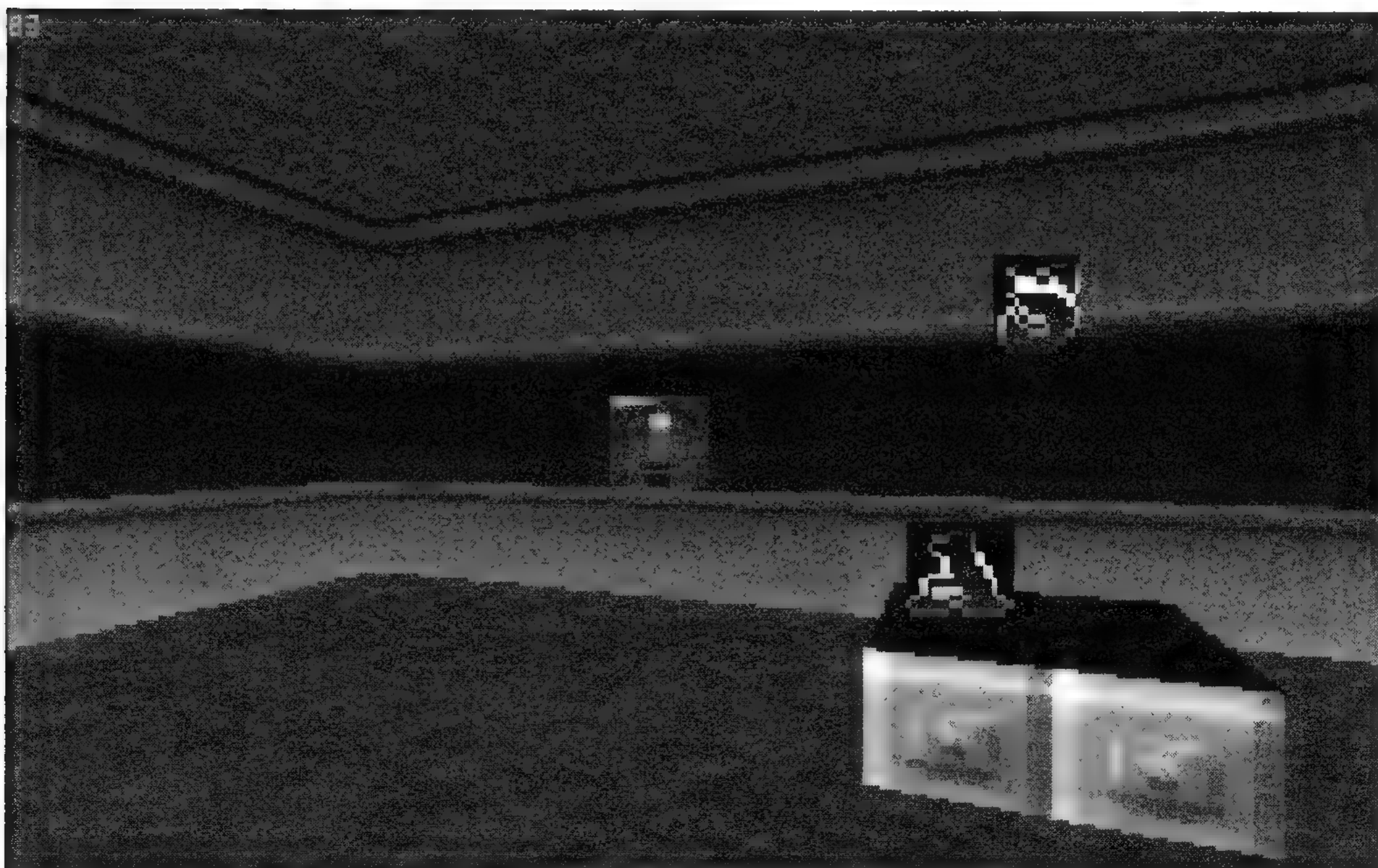


FIGURE 7.50: In this 3D view of the moving floor effect, the SectorEffector's height is different from the current floor height, so that the floor will move when the switch is flipped.

Next, switch to 3D view mode, and change the height of the SectorEffector so that it's different from the current ceiling height. Return to 2D view mode, and decide how you want the ceiling to move:

- ❖ Make the ceiling move *from* its current height *to* the height of the SectorEffector, with the SectorEffector's angle pointing upward (north), or
- ❖ Make the floor move *from* the height of the SectorEffector *to* the designed height, with the SectorEffector's angle pointing downward (south).

However, the angle of SE 32 works opposite the way it does for SE 31. An *upward* angle will start the sector's ceiling at its designed height, and it will go to the SectorEffector's height when activated. If the angle is *downward*, the ceiling will start at the SectorEffector height and travel to the designed height.

For this effect, set up the Activator and Switch sprites as you did for the floor move effect to trigger the ceiling move. Figure 7.51 shows a 3D view of this effect.

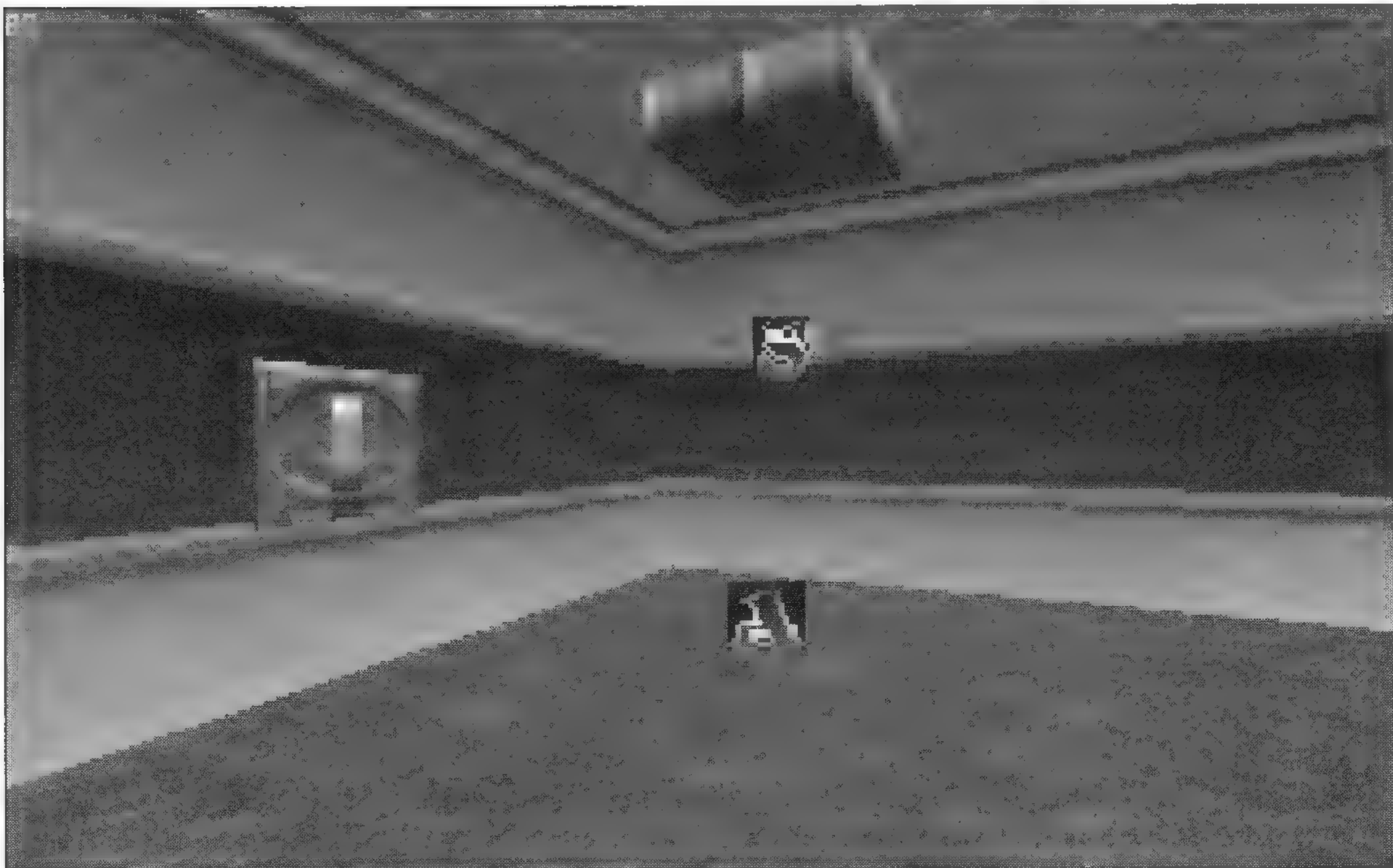


FIGURE 7.51: This is a 3D view of the ceiling move effect with SE 32.

EARTHQUAKE JIBS (33)

Difficulty: Easy

This SectorEffector will spawn debris from the earthquake jib locations whenever an earthquake is triggered on your map. Any earthquake will trigger all of the SE 33s you've placed on the map, so there is no need to tie them to the earthquakes with LoTag or HiTag values.

To use this sprite, simply place a SectorEffector anywhere on your map, and give it a LoTag value of 33. The height of the SectorEffector is important as well. An SE 33 on the ceiling will cause the debris to fall from that spot, while an SE 33 on a floor will cause the debris to roll around on the floor.

SHRINK RAY SHOOTER (SE 36)

Difficulty: Medium

SectorEffector 36 will create some type of projectile every 5 seconds after being activated by a MasterSwitch or an Activator. The first example of this sprite is seen in the Toxic Dump level in L.A. Meltdown (E1L4), in the room with the crane. The shooter fires a shrink ray at the player, and then the shrunk player must travel through very small tunnels to get to a switch.

To create a shooter, make a room sector and a small sector off to one side. Raise the floor and lower the ceiling of the small sector so that it becomes a small indentation in the wall, similar to what you see in Figure 7.52. Place a SectorEffector in the small sector, and give it a LoTag value of 36. Point the SE 36's angle in the direction you want the projectile to shoot. Also, place an Activator sprite in the small sector, and give it a unique LoTag value (x). Finally, place a GPSpeed sprite in the small sector, and set its LoTag value to 2556. This number represents the projectile fired from the shooter.



TIP

The projectiles listed in Table 7.1 are included in the Build documentation. However, other projectiles seem to be possible. I tried COOLEXPLOSION1 (1360), which is the octabrain ammo, as well as FREEZEEXPLOSION (1641), and they both shot! However, the DEVISTATOREXPLOSION (1642) did not fire. Feel free to embark on your own experimentation here.

Now, place a Switch sprite somewhere in the big room, and give it a LoTag value of x. This should activate the shooter. You can now go and test it in the game. If you want to change what the shooter shoots, you can change the LoTag value of the GPSpeed sprite according to Table 7.1.

TABLE 7.1: GPSPEED SPRITE LOTAG VALUES FOR CHANGING PROJECTILES

| LOTAG VALUE | PROJECTILE |
|-------------|--|
| 2556 | Shrink ray |
| 2605 | RPG |
| 1650 | Mortar (L.A. Meltdown boss shoots these) |
| 1625 | Trooper laser |
| 1636 | Lizardman spit (from Assault Troopers) |



QUICK FIXES

The Build documentation states that the shrink ray shooter effect will never stop when it is started and that it can be activated only by a MasterSwitch sprite. However, the exercise in this section uses an Activator sprite and a Switch sprite to start the shooter, and turning off the switch *does* stop the shooter. Now you know something that maybe even the *Duke Nukem 3D* level designers don't know!

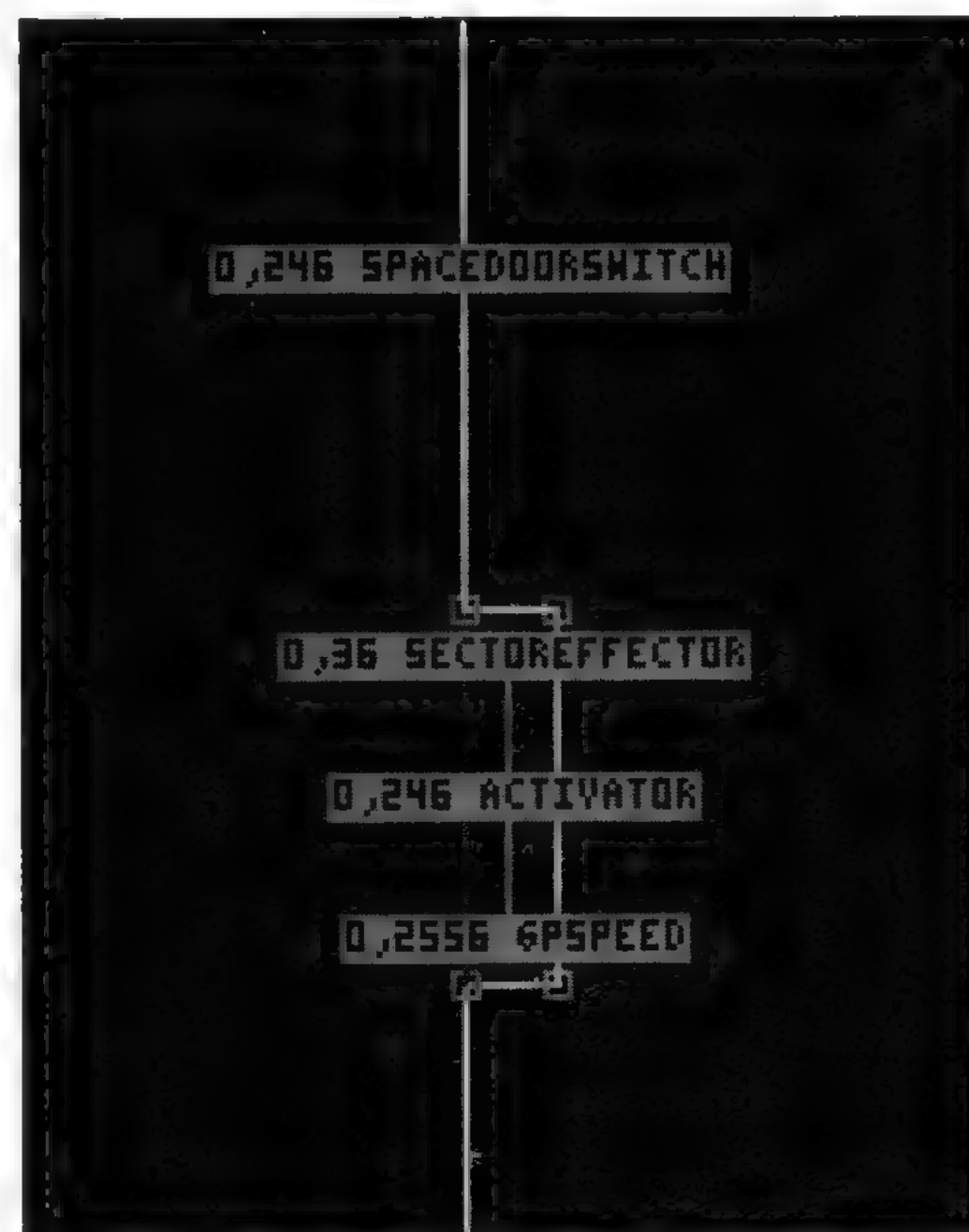
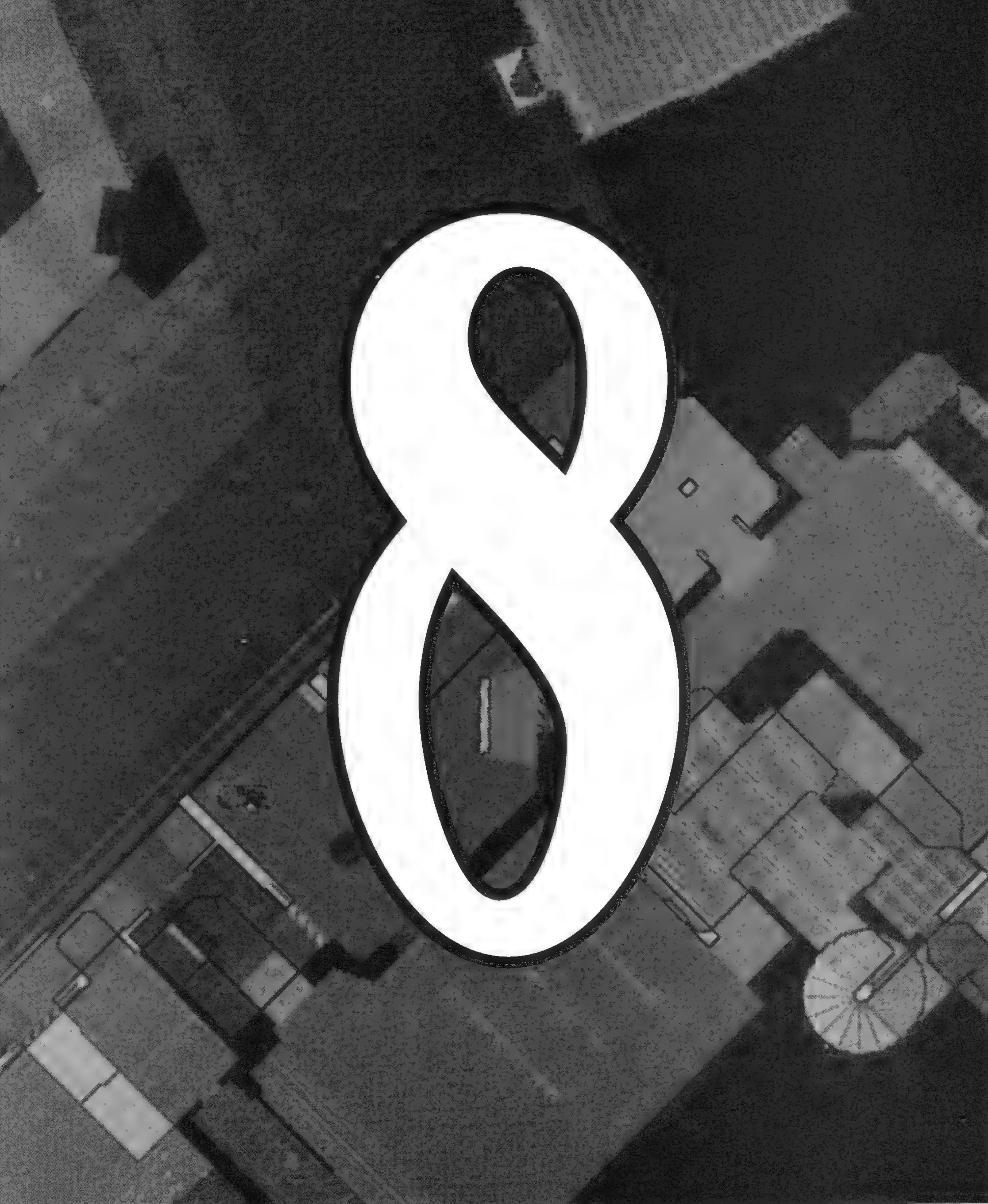


FIGURE 7.52: Set up the smaller sector along the wall of the larger sector to set up a shooter effect.





Special Construct Effects

The prior chapter dealt with all the special effects that can be created using sectors. This chapter will cover the other *constructs*, most of which use either a sprite or a single wall to achieve a special effect. As you work through this chapter, you will see how to modify your sectors' walls to create effects that can really make your levels extraordinary.

WALL CONSTRUCTS

The realism of the *Duke Nukem 3D* environment is in large part due to the simple enhancing touches that may go unnoticed by most players. Effects like transparent windowpanes that provide vistas of outer space or mirrors that reflect just how good Duke really looks while in the heat of battle are what makes this game's 3D environment so appealing.

CREATING MASKED WALLS

Normally, a two-sided wall is transparent and has two possible areas in which a texture can be applied—the lower section and the upper section. Consider a windowsill sector that has both a lower ceiling and a higher floor than the sector the player is standing in. The area between the ceiling and floor will normally be transparent, allowing the player to view through the wall to the other side of the sector, and the outward-facing wall surfaces below the floor and above the ceiling will have a texture, as shown in Figure 8.1.

In special cases, however, it may be desirable to have a texture *between* the floor and ceiling as well. Consider again the window sector above, but suppose the designer

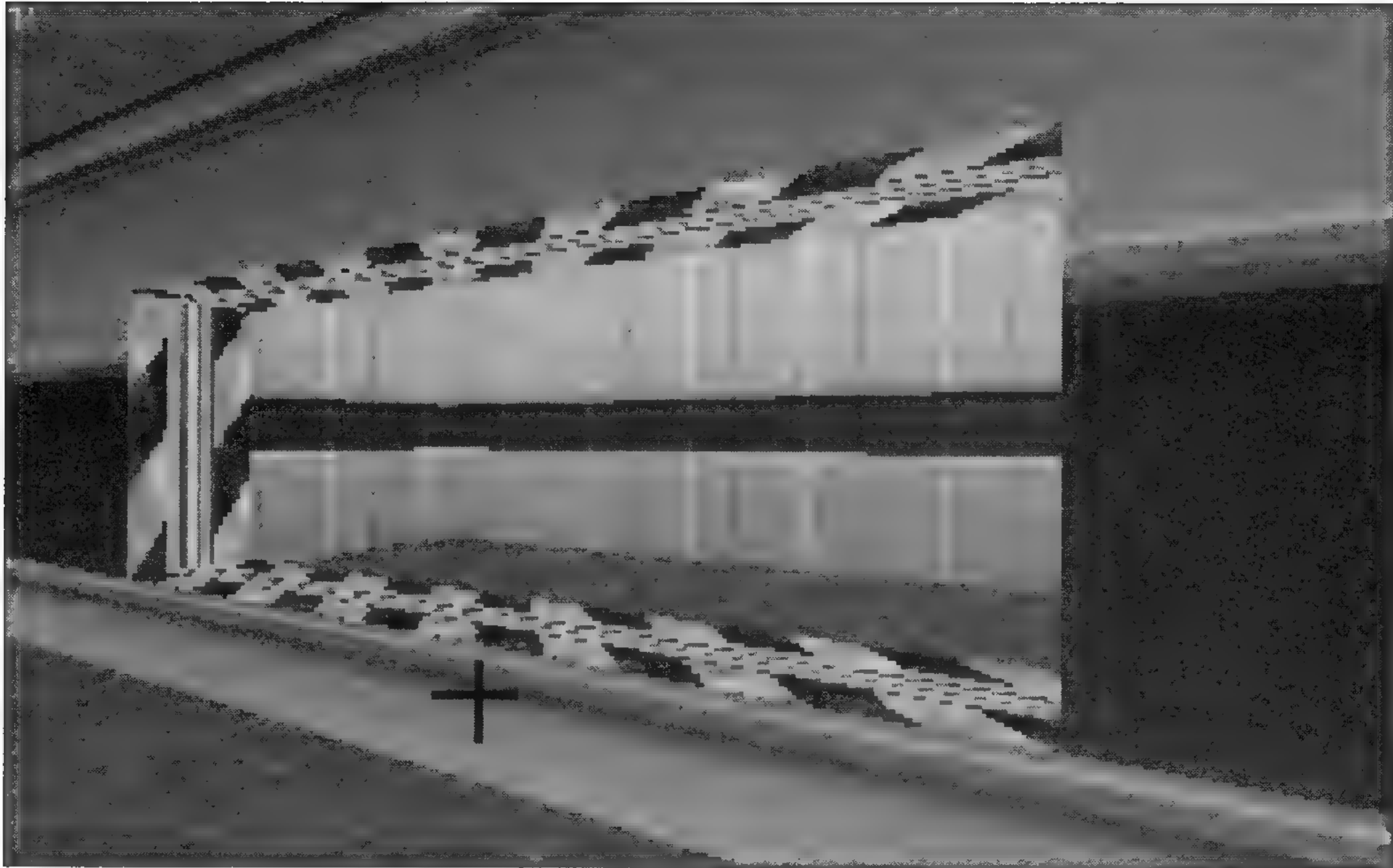


FIGURE 8.1: This *normal* two-sided wall can be made into a masked wall by placing the mouse cursor where you see the dark +.

put some glass in the window. The glass is a texture that will have to be put on the transparent wall. To place a wall texture on what would normally be a transparent area on a two-sided wall, you need to create what's known as a *masked wall*. Masked walls are used to create glass, mirrors, and force fields.

To create a masked wall, first construct the sectors. Then, while in 3D view mode, place the mouse cursor on the base of the floor sector where you would like the wall and press the M key (the area to position the mouse cursor is shown in Figure 8.1). The area between the ceiling and floor will fill with the default brown brick texture. You can then change the texture as you wish.

Creating masked walls is a necessary skill that you will need to create some of the construct effects that follow.

Glass Walls

The first type of special wall you can create is a glass wall that breaks when shot out. To create a glass wall, first make three adjoining sectors, with the middle one much thinner than the outer two (see Figure 8.2). This middle sector will be a window.

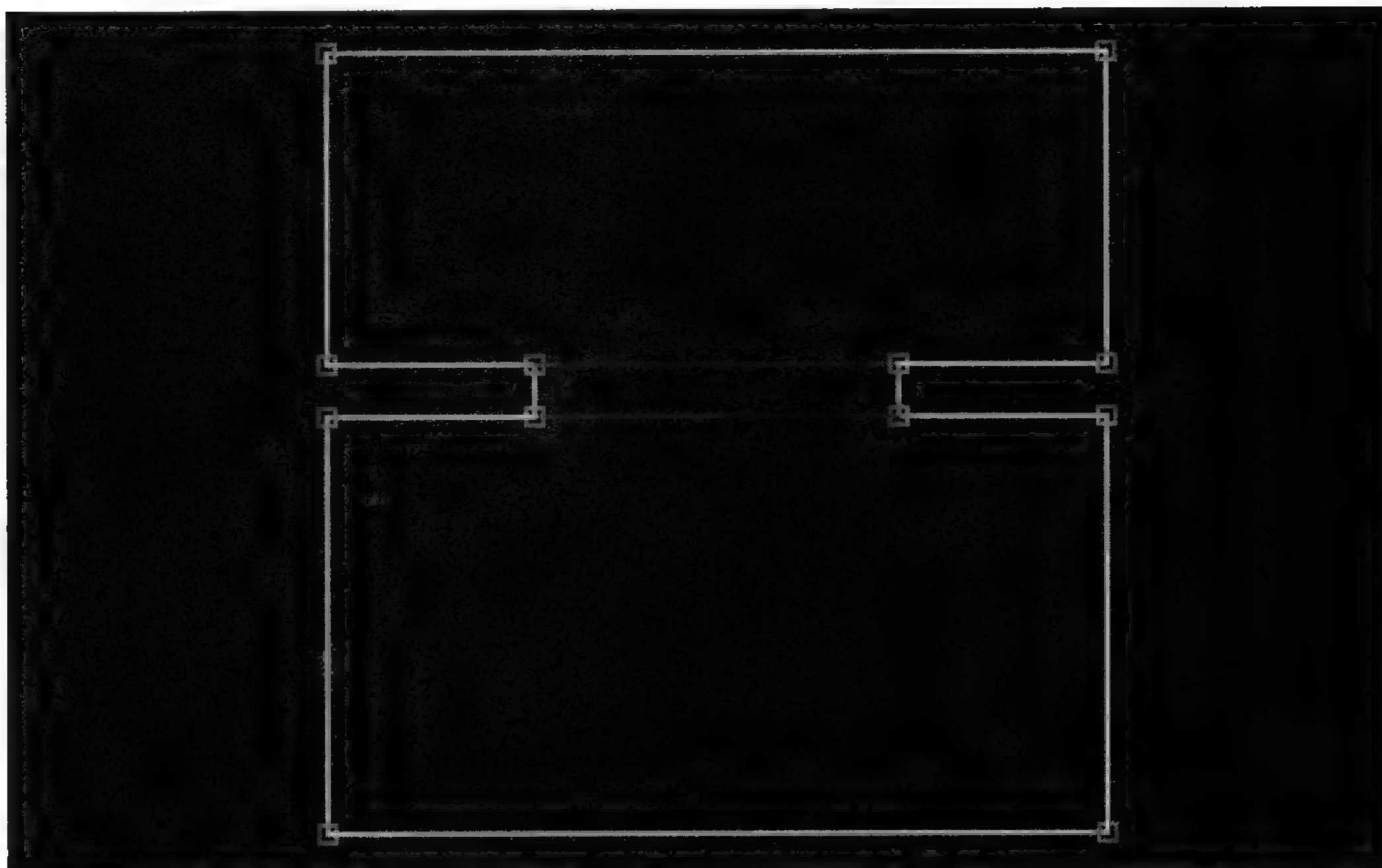


FIGURE 8.2: This is the sector layout for setting up a glass wall or a window in a masked wall.

Switch to 3D view mode, and lower the floor and raise the ceiling of the window sector to create a windowsill. Your windowsill is now ready for some glass. Place the mouse cursor at the base of the windowsill, right where it meets the floor (see Figure 8.1 for the location). Press the M key to mask this wall. This creates a new texture between the upper and lower sections of the wall. The texture that appears is the default brown brick texture. The glass texture is #503. Change the new masked texture to this number the same way you change any other texture.

Now, place the mouse cursor on the glass wall and press the B key. This makes the wall *blockable*, which means that objects won't be able to pass through it. In this case the B key is a toggle key; press it a second time to remove the Blocking bit. Then, press the H key to turn on the HitScan bit. The HitScan bit allows the wall (and window) to be shot out. Finally, press the T key either once or twice, to make the wall either 25 percent or 50 percent transparent. Your glass wall should be ready for breaking.

In 2D view mode, a wall that's marked with a Blocking bit will turn purple in color, while a wall with the HitScan bit will appear thicker than the other walls. Your glass wall here should be both thick and purple because it is now set with both the Blocking and HitScan bits turned on.

You can also set the Blocking and HitScan bits in 2D view mode. The Blocking bit is still set using the B key, but the HitScan bit is set using Ctrl + H instead of the H key alone,

as it is in 3D view mode. Practice turning on and off the Blocking and HitScan bits a few times before moving on to the next section. Figure 8.3 shows a 3D view of the final glass wall effect.

Force Fields

A force field is also created by using a masked wall. To create one, create a masked wall as you did for the glass wall, but use texture #663 for the force field, and you don't need to set the HitScan bit. If you do turn on the HitScan bit, the wall will block bullets even when the force field is off. Figure 8.4 shows a 2D view of a switch-controlled force field.

If you would like an invisible force field, you can unmask the wall after applying the force field texture. The wall will still act as if a force field is present even though you won't be



TIP

Don't go too crazy with transparent wall textures on masked walls, because they slow down the frame rate dramatically. As a test, spin your viewpoint around in a circle by holding down the right or left arrow key, and watch the frame rate in the upper left corner of the screen as the glass wall comes into view, then leaves the view. You should see a demonstrable decrease as you pass the glass wall. On my machine, the frame rate dropped from 70+ down to around 55–60 with *only* the three sectors you see in the example!

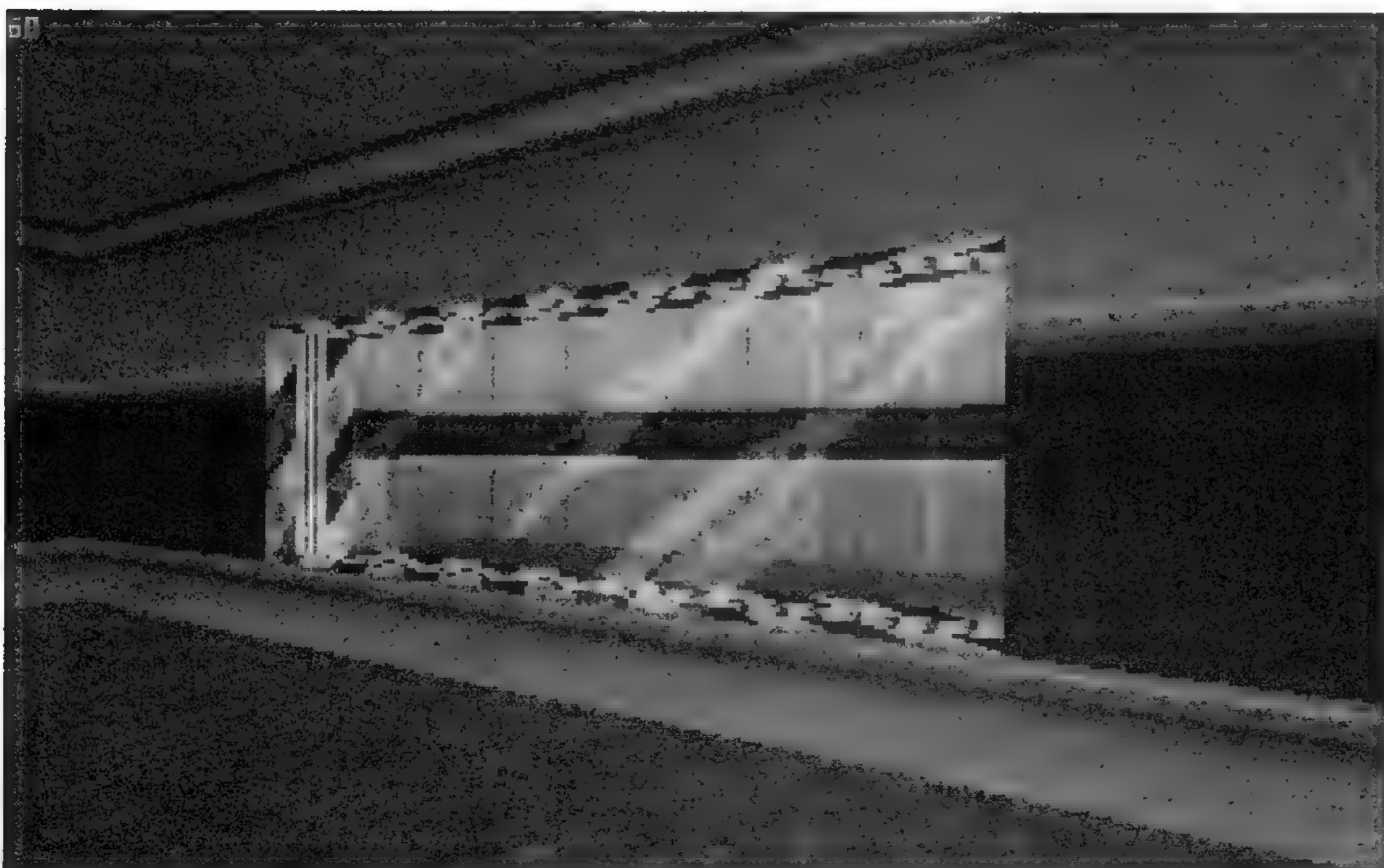


FIGURE 8.3: Here's the final glass wall effect in 3D view mode.

able to see it. To make the force field controllable by a switch, give the force field wall a unique LoTag value. (Recall that the label for a wall's LoTag and HiTag values is red; don't accidentally give a sector or a sprite the same LoTag value.) Then, place a Switch sprite somewhere in your level, and give it the same LoTag value.

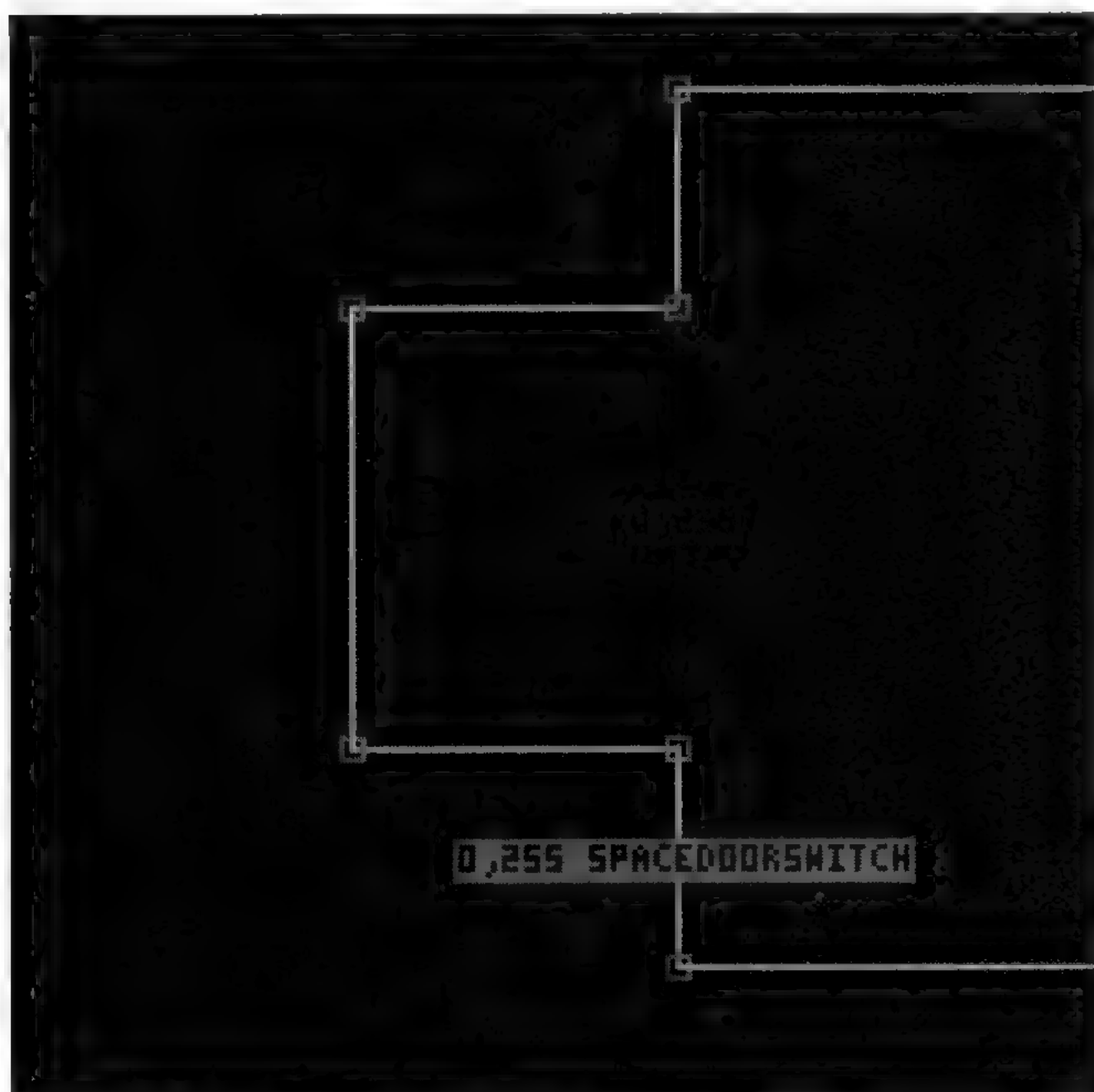


FIGURE 8.4: This 2D view of a sector layout creates a force field that is controlled with a switch.

Mirrors

The mirror is another special effect of the Duke engine that is created from a masked wall. This effect requires creating an extra sector that acts as the *reflecting area*. Figure 8.5 shows a scene shot from the Hollywood Holocaust level (E1L1) loaded into Build, which shows a large room behind the mirrored wall. This room is required to create a proper reflection in the mirror.

Begin by creating the sector that you want the player to be standing in when he or she is looking in the mirror. Create a wall that will be the mirror wall. Then, place another sector behind this wall so that it looks something like the one shown in Figure 8.5. Now, switch to 3D view mode, and mask the mirrored wall as you did for the glass wall above. The mirror texture is #560. Turn on the Blocking and HitScan bits for this wall using the B and H keys, respectively. Finally, make this a one-way wall by pressing the 1 key. When

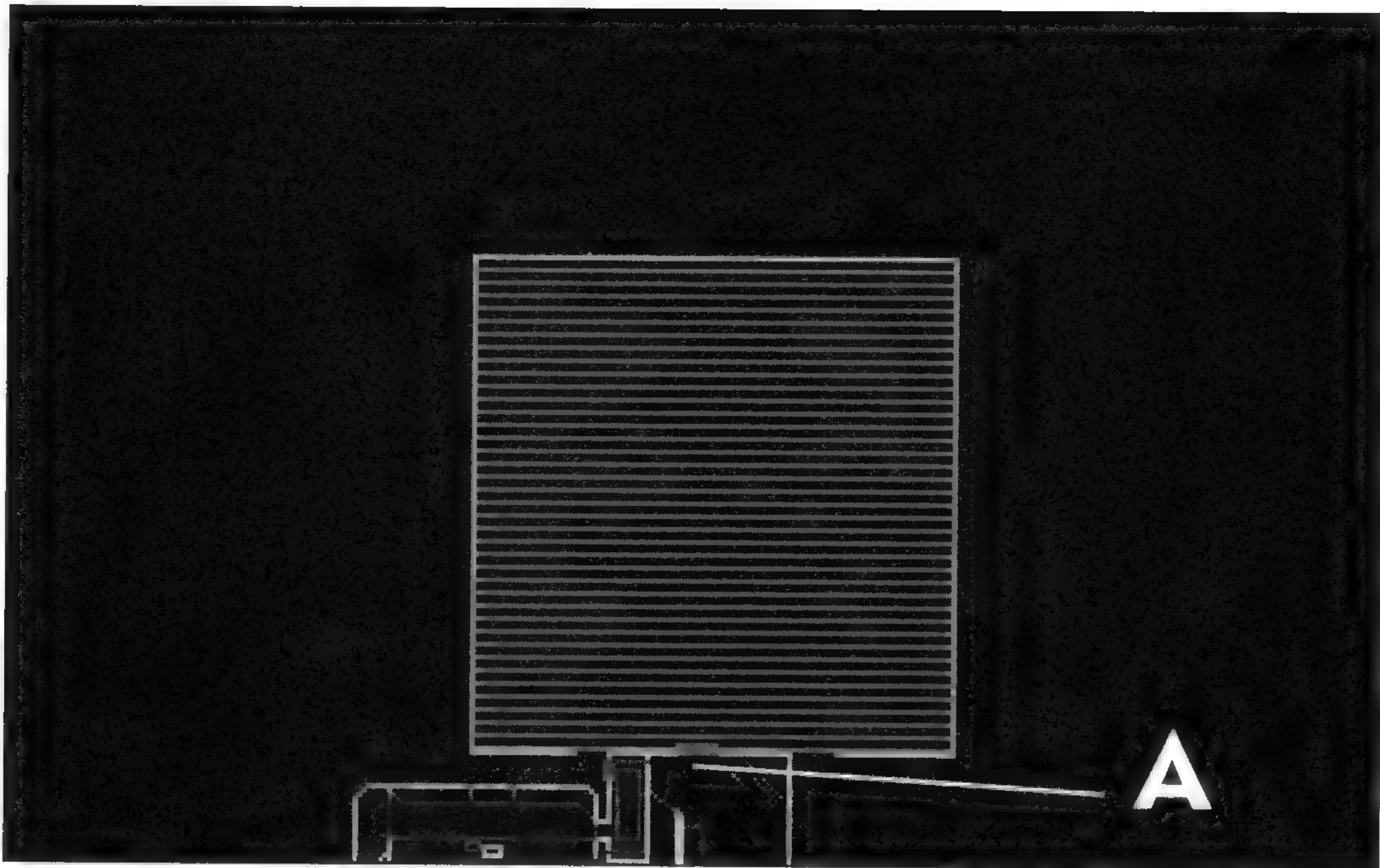


FIGURE 8.5: Mirrors require a large sector behind them to act as a reflecting area. The reflecting area is shown on your screen in green (the striped area). The size of this sector will control the reflection in the mirror. The light-colored line shown at position A is the mirror.

you do this, you will see an opaque magenta color where all the reflecting parts of the mirror should be, as shown in Figure 8.6.

When you try out your mirror in the game, if you see strange reflective effects when standing at certain angles to the mirror, try adjusting the width and length of the room behind the mirror. The room should be wider than the length of the mirrored wall and longer than the room that the mirror faces.

If you want Duke to say, “Damn, I’m looking good,” when the player hits the spacebar at your mirror, assign the LoTag of the mirror wall the value 252, which is the sound number that will play.

SPRITE CONSTRUCTS

In addition to special types of wall constructs, there are certain constructs created by using only sprites. These special types of sprites allow for additional player interaction, which all helps to make your *Duke Nukem 3D* level seem like a more realistic place.

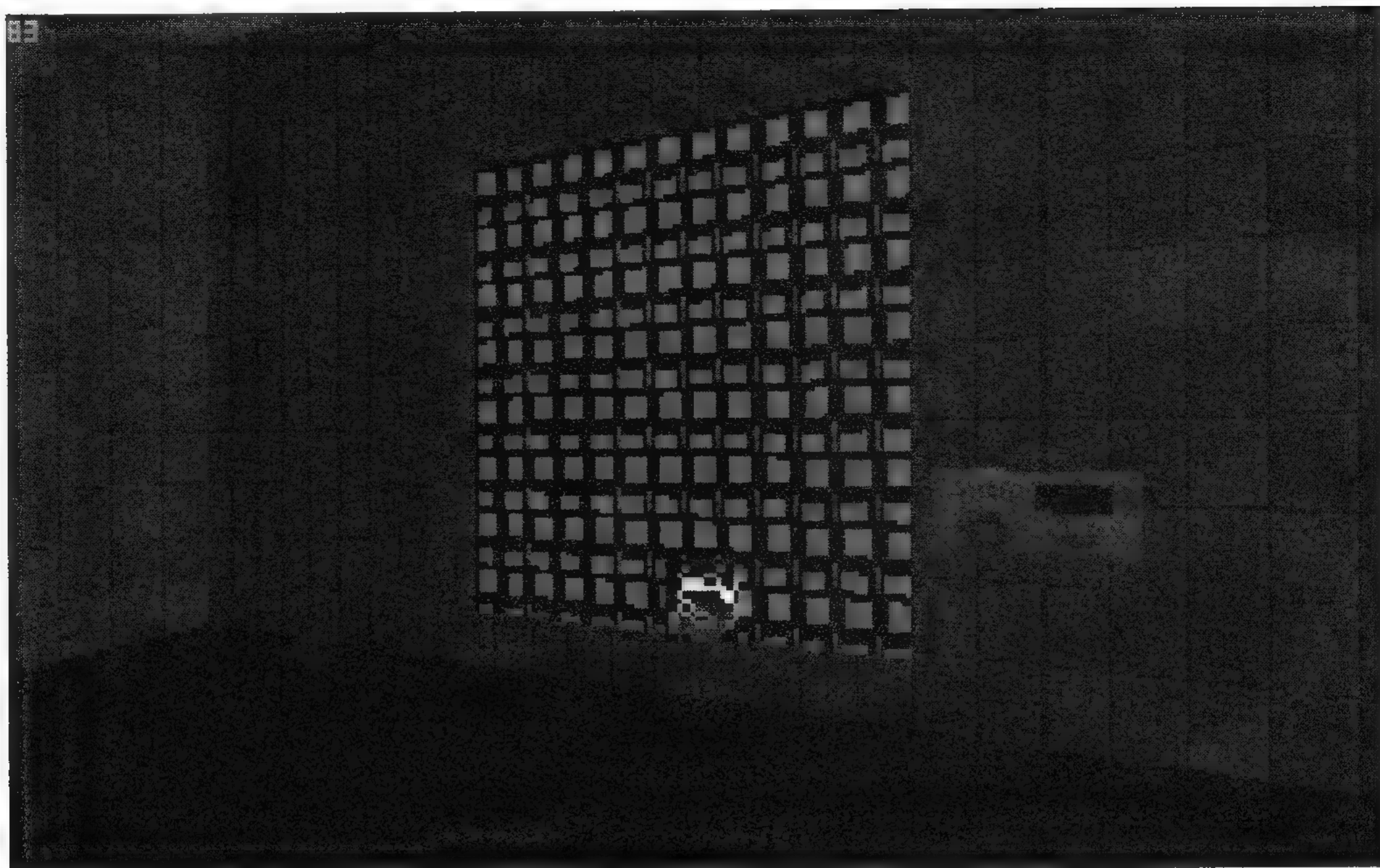


FIGURE 8.6: The mirror in 3D mode won't appear to reflect properly until you try it out in the game.

VENTS

The breakable vents that block air ducts are created by using a sprite. To create one, first create the sectors that will be the air duct and the room the vent faces, as shown in Figure 8.7.

While you are in 2D view mode, place a sprite on the wall that joins these two sectors. Make the angle of the sprite face perpendicular to the wall. You can automatically place the sprite against the wall and adjust its angle using the O key, or you can do it manually. Once the sprite is in place, switch to 3D view mode, and give the sprite texture #595 (GRATE1). Also, use the R key to make sure the sprite will lie flat against the wall, instead of always facing the player. You can also stretch the vent texture to fit your air duct opening.

Finally, make sure to go inside the air duct and see if there's a vent texture on the other side of the sprite. If not, put the mouse cursor where the sprite should be and press the 1 key to make the sprite visible from both sides. Figure 8.8 shows a 3D view of the final breakable vent effect.

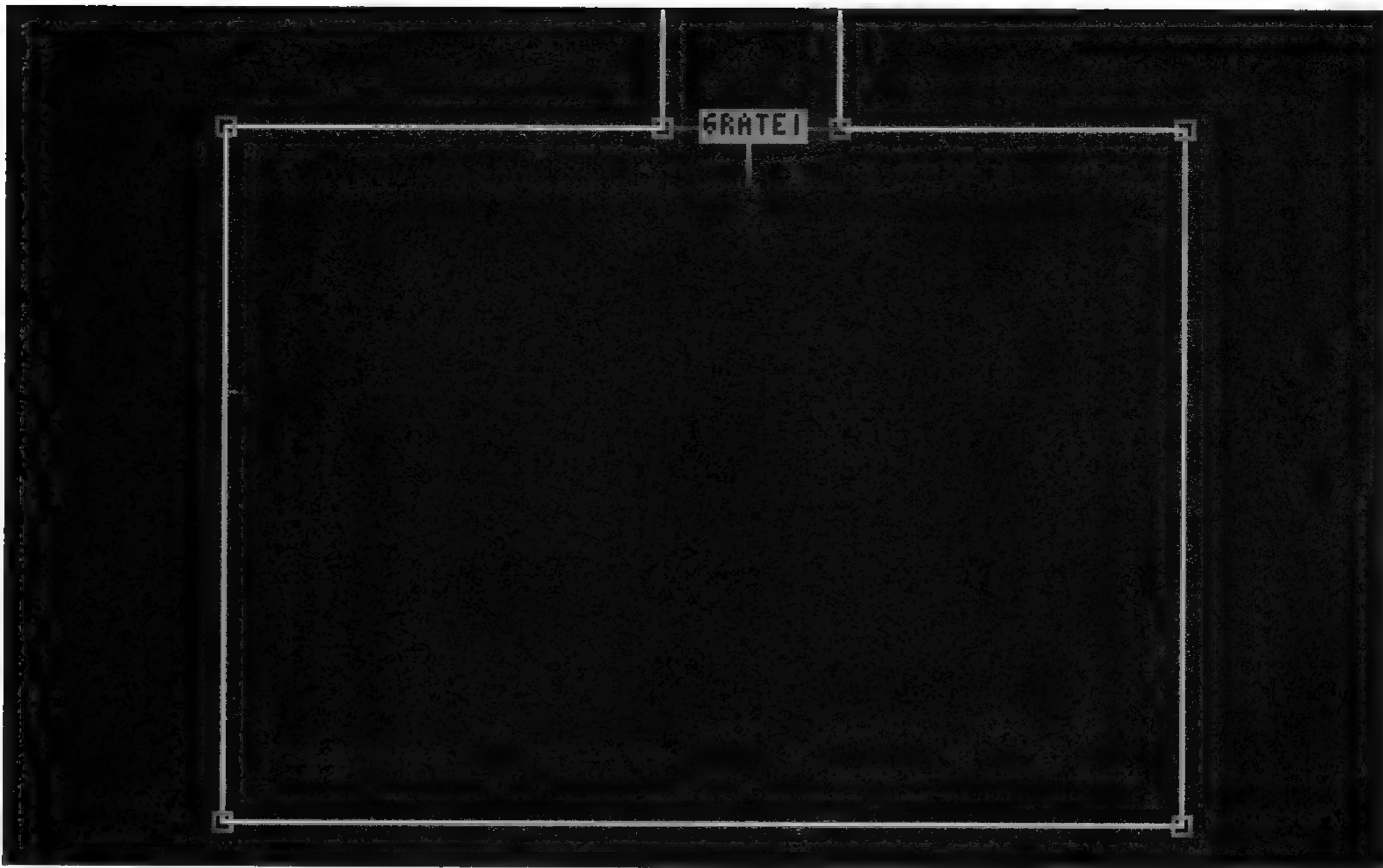


FIGURE 8.7: These sectors are set up to create an air duct behind a breakable vent.

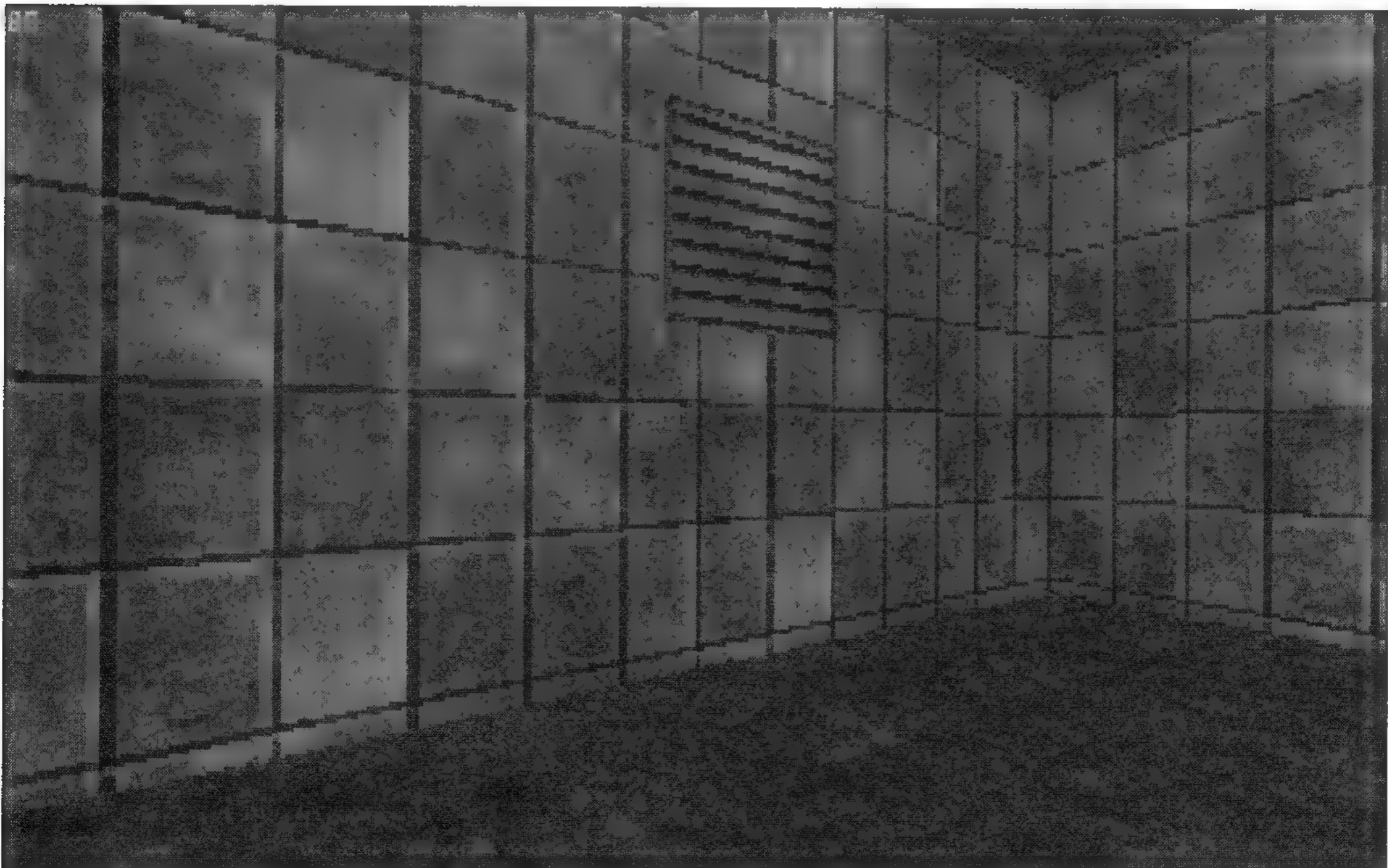


FIGURE 8.8: Here's the completed breakable vent effect in 3D view mode.

SECURITY CAMERAS AND MONITORS

The ViewScreen sprite (for the security viewing monitors) is #502. Place one of these anywhere flat against a wall. Give this sprite a unique HiTag value. Then, place a Camera sprite (texture #621) somewhere else on the map. Assign the same value you used for the ViewScreen sprite's HiTag value for the Camera sprite's LoTag value. (See Figure 8.10.)

The angle of the sprite controls the angle that the camera faces. You can make the camera pan back and forth by giving it a HiTag value, which should be expressed in Duke Nukem degrees, that is, 2048 equals 360 degrees. For example, if you assign the camera a HiTag value of 512, the camera will pan 90 degrees—45 degrees to the left of the start angle and 45 degrees to the right. Finally, to aim the camera downward, set the *shade* of the camera.

With this kind of setup, all the cameras with the same LoTag value (matching the ViewScreen sprite's HiTag value) will be viewed through the same ViewScreen consecutively. Figure 8.11 shows a 3D view of the security monitor effect.

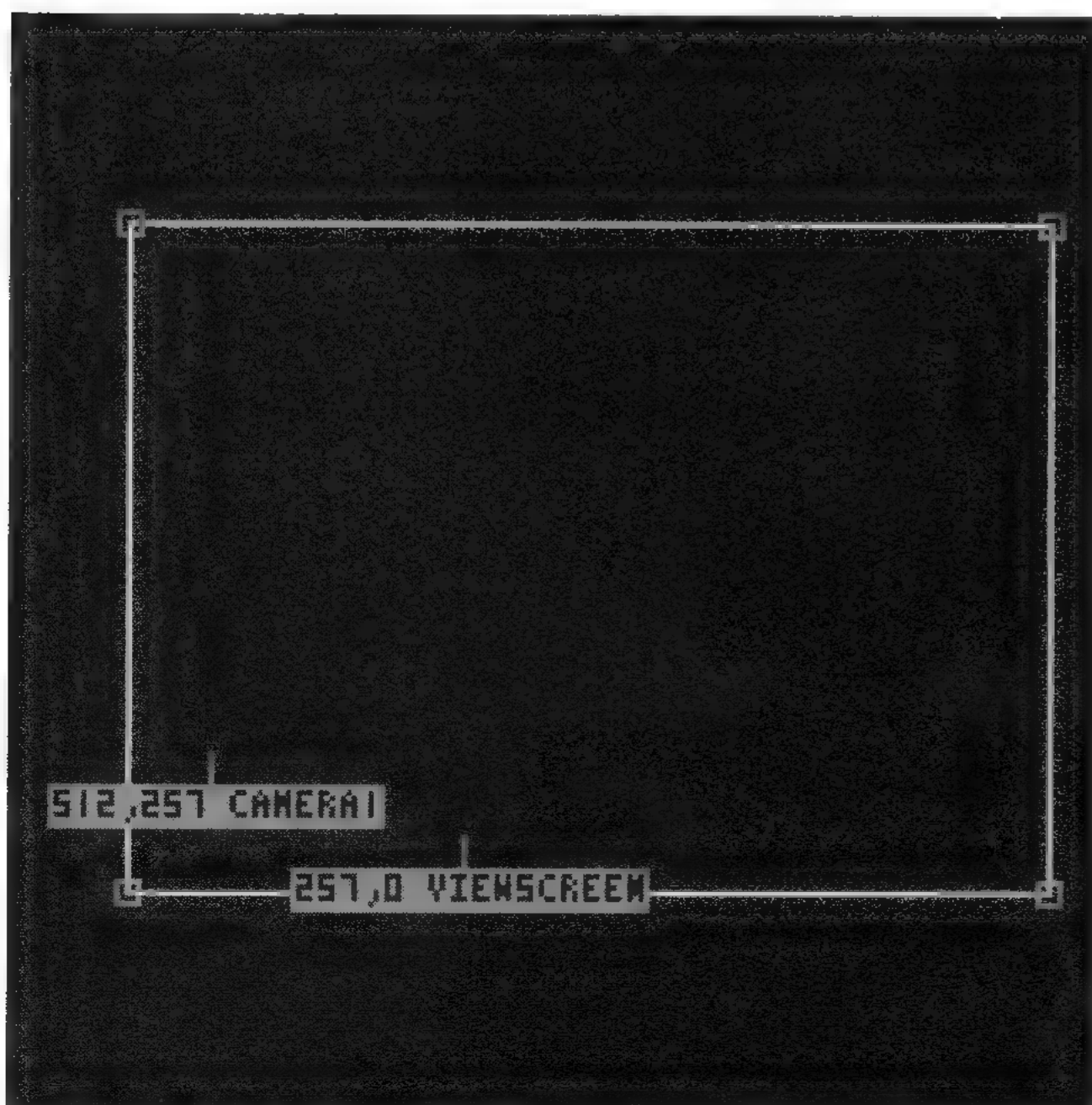
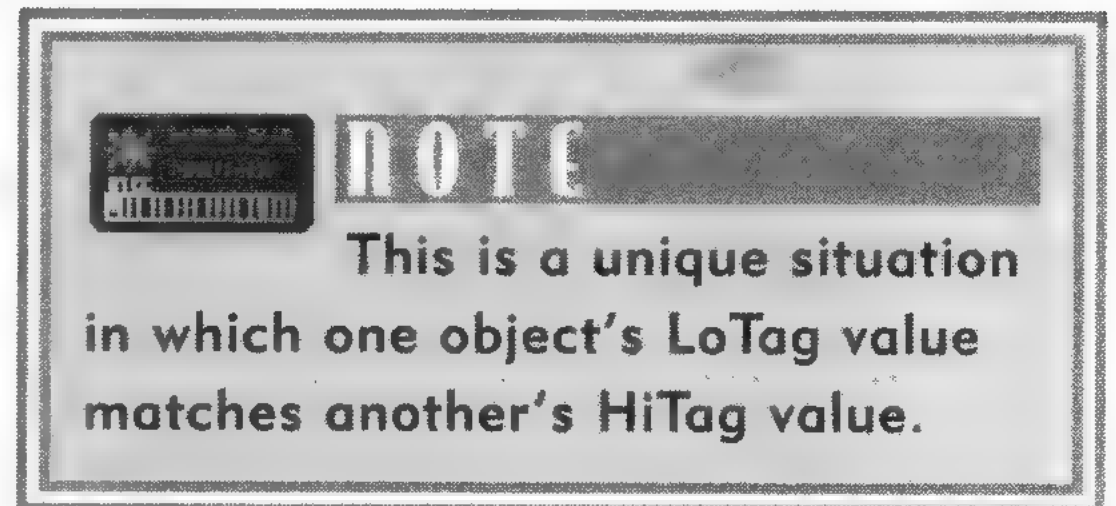


FIGURE 8.10: Here is a typical arrangement for a Camera sprite and a ViewScreen sprite.



FIGURE 8.11: The Camera and ViewScreen sprite pair shown in Figure 8.10 produces the effect you see here.

KEYS AND LOCKED DOORS

Doors that require a keycard are created using the AccessSwitch sprite (130). Start by creating a normal closed door or any other type of door you like (sliding, *Doom*-style, Star Trek, etc.). Place an ActivatorLocked sprite (4) inside the door sector, as shown in Figure 8.12. Give the ActivatorLocked sprite a unique LoTag value. Then, place the AccessSwitch sprite on the wall nearby, and give it the same value for its LoTag field, and assign its HiTag a value of 212 (which is the door unlocking sound effect).

Leaving the palette of the AccessSwitch 0 will make a door that requires a blue keycard. To make a door require a different color keycard, change the palette of the AccessSwitch sprite to 21 to require a red key, or 23 to require a yellow key.

When you place a Key sprite (60) somewhere on your map, change the color of the key to the same palette as the AccessSwitch sprite by using the same values.

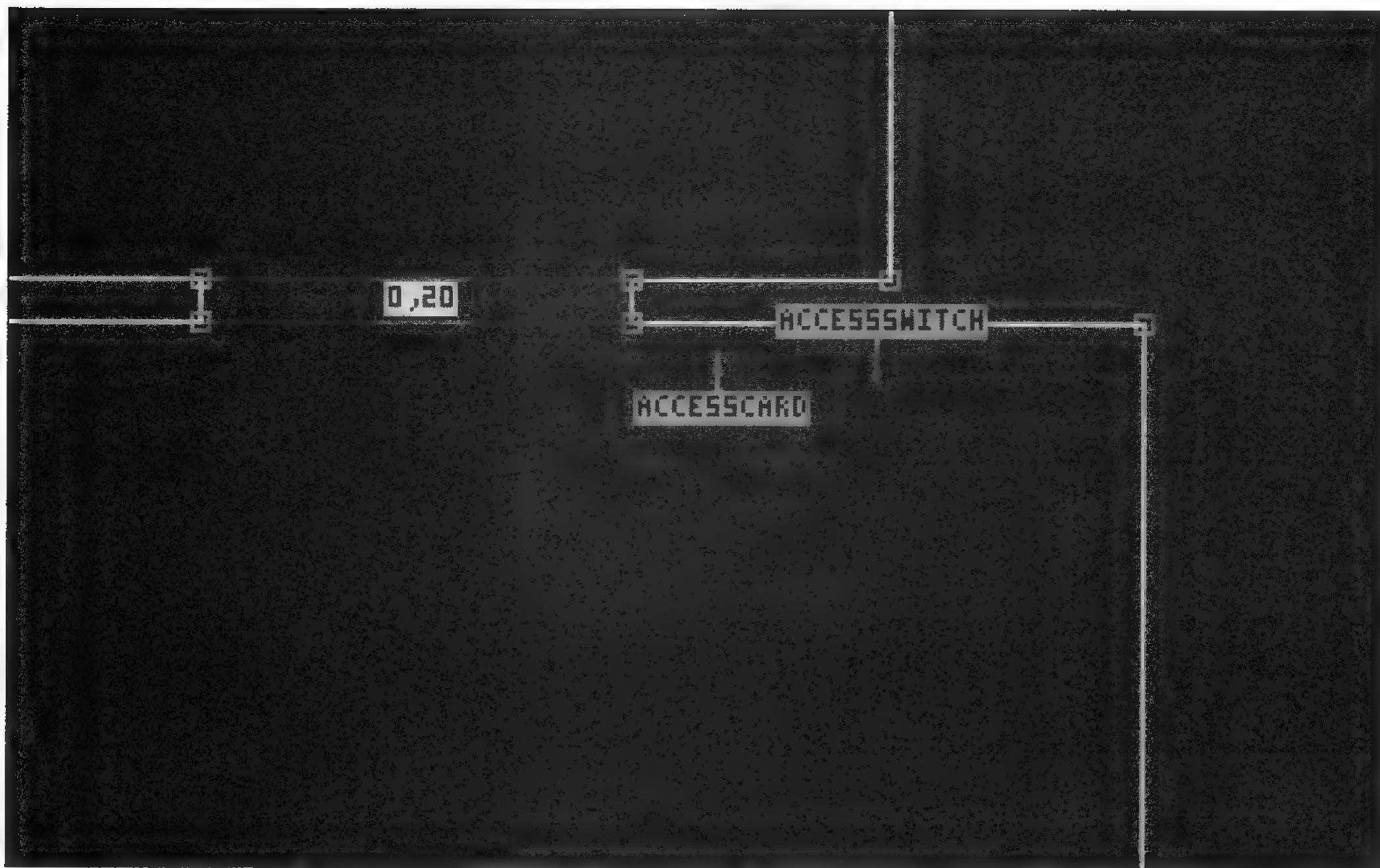


FIGURE 8.12: This is an arrangement of Key and AccessSwitch sprites for a locked door and keycard.

MISCELLANEOUS CONSTRUCTS

There are a few miscellaneous constructs that require neither special walls nor sprites, but are a bit tricky to create nonetheless. These are described here.

SPACE

One effect that can be quite *out of this world* is a window that looks out into the void of space. However, it is a fairly tricky process, as follows.

Begin by drawing a sector that will act as the place the player will stand to look out the window. Call this the *view* sector. Next, create a thin sector that follows the room sector along one wall. Call this the *sill* sector. Adjust the floor and ceiling of the sill sector so that it looks like a windowsill.

Create a large sector on the other side of the sill sector, called the *space* sector. Make it wide so the player can turn left and right and still be looking out into space. Make sure the space sector has a higher ceiling than the view sector. Then, make a thin sector

that borders the space sector on all sides except the sill sector's side. This will be the *edge* sector. Make the floor and ceiling of the edge sector equal to those of the space sector.

Next, apply the texture BIGORBIT1 (or any of the other BIGORBIT textures) to the floor and ceiling of the space sector. Parallax the floor and ceiling of the texture by pressing the P key. Do the same for the edge sector, that is, apply the BIGORBIT1 texture to the floor and ceiling and parallax them.

Finally, raise the floor of the edge sector upward so it meets the ceiling. This will make the entire area parallax for what looks like eternity. Figure 8.13 illustrates this construction. Recall that any sector with the BIGORBIT textures will kill Duke, even if he is in God mode. To get around this, you can put a palette on the floor and ceiling of the space sector. Use palette 3, because it has no visible effect on the textures.

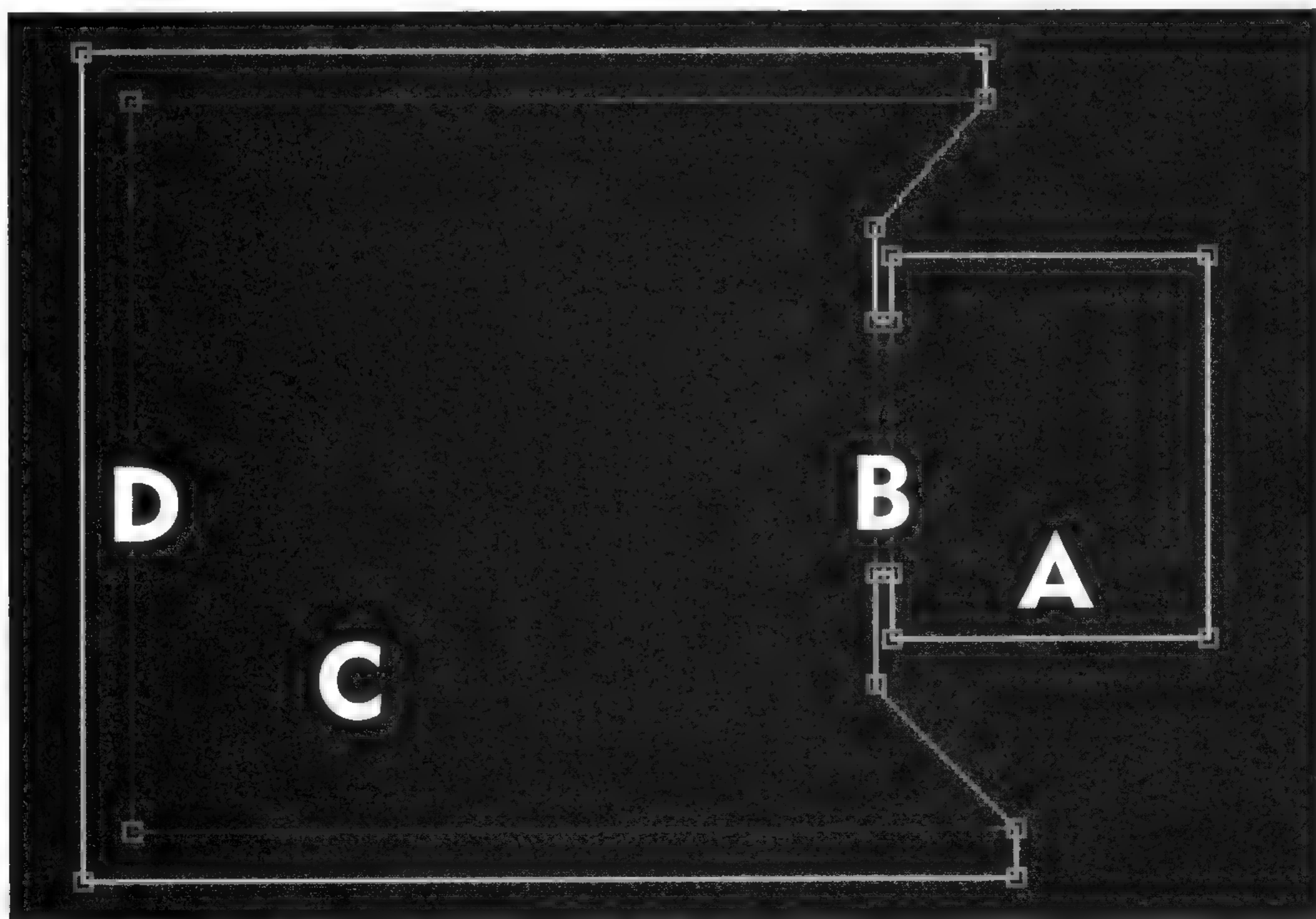


FIGURE 8.13: Together the view sector (A), the sill sector (B), the space sector (C), and the edge sector (D) create the effect of looking through a window into the void of outer space.

LOW PERIMETER WALLS

If you have an outdoor area on your map, most of the time the ceiling of the outdoor area will be very high, so that you have room to put all the buildings and structures

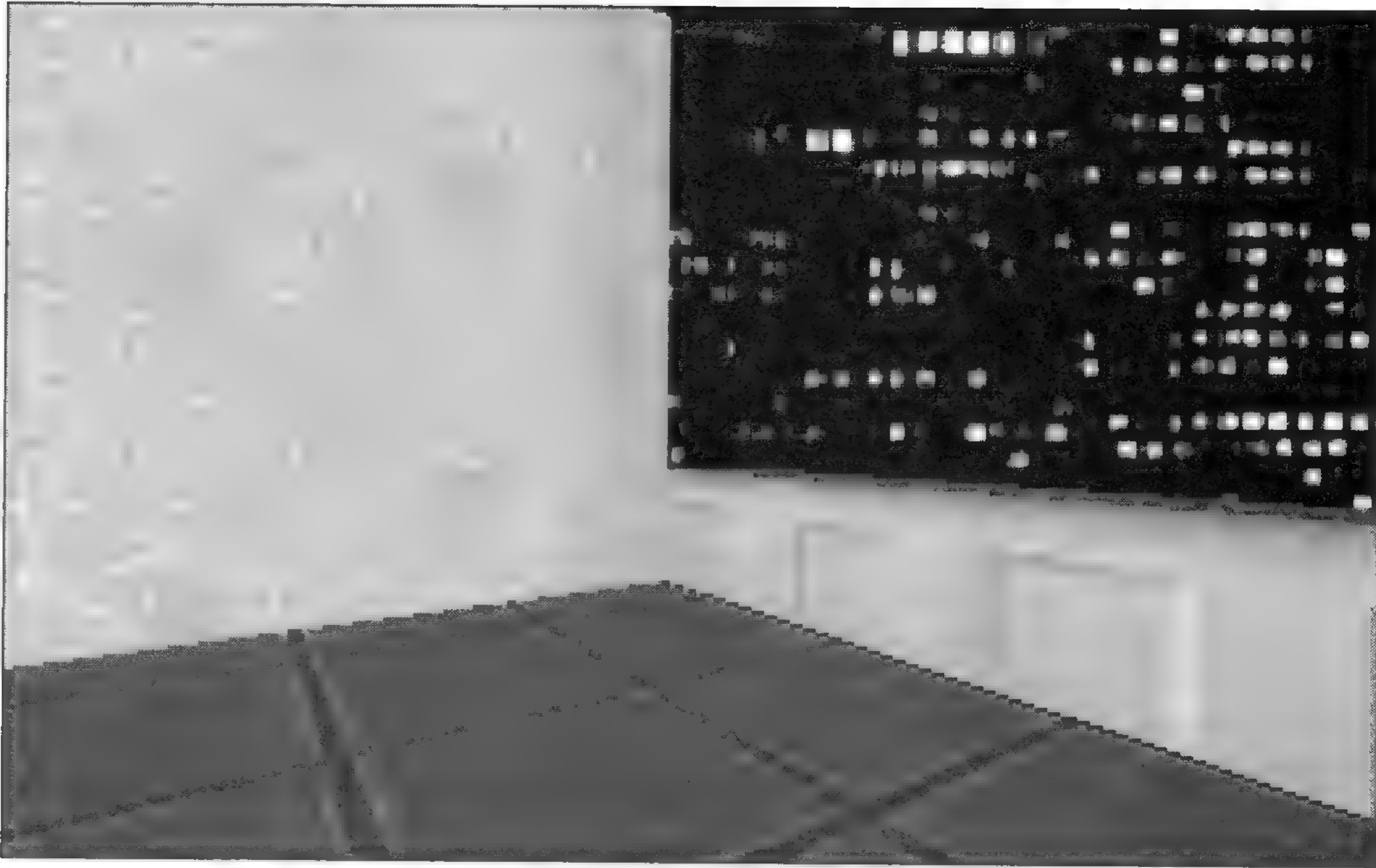


FIGURE 8.14: The low perimeter wall, to the right, is many times a better effect to border a level than the tall blank wall to the left.

into the large outdoor sector. Normally, having this tall sector would mean the edges of the outdoor sector would require very tall walls going all the way up to the ceiling of the sector. However, these tall walls will, many times, prevent the player from being able to see the parallaxed sky texture on the ceiling.

It may be preferable to have the perimeter of your outdoor area a low wall, similar to the one shown in Figure 8.14. This allows the player to see the parallaxed sky texture, and it gives the level a much more *open* feeling.

Start this effect by making three parallel sectors at the edge of the area you'd like to surround with a blockable low perimeter wall, similar to the arrangement shown in Figure 8.15. Note the labels A, B, and C for the sectors. These sectors will be referred to by their letters in the directions that follow.

Sector A will act as the actual perimeter wall. Raise the floor of this sector up a few clicks to be the desired wall height. Also, give the ceiling of this sector the parallaxed sky texture that you want. Sector B should be given a parallaxed ceiling. It is unimportant what the texture of the floor or side walls of this sector are; they will be hidden when the effect is complete.

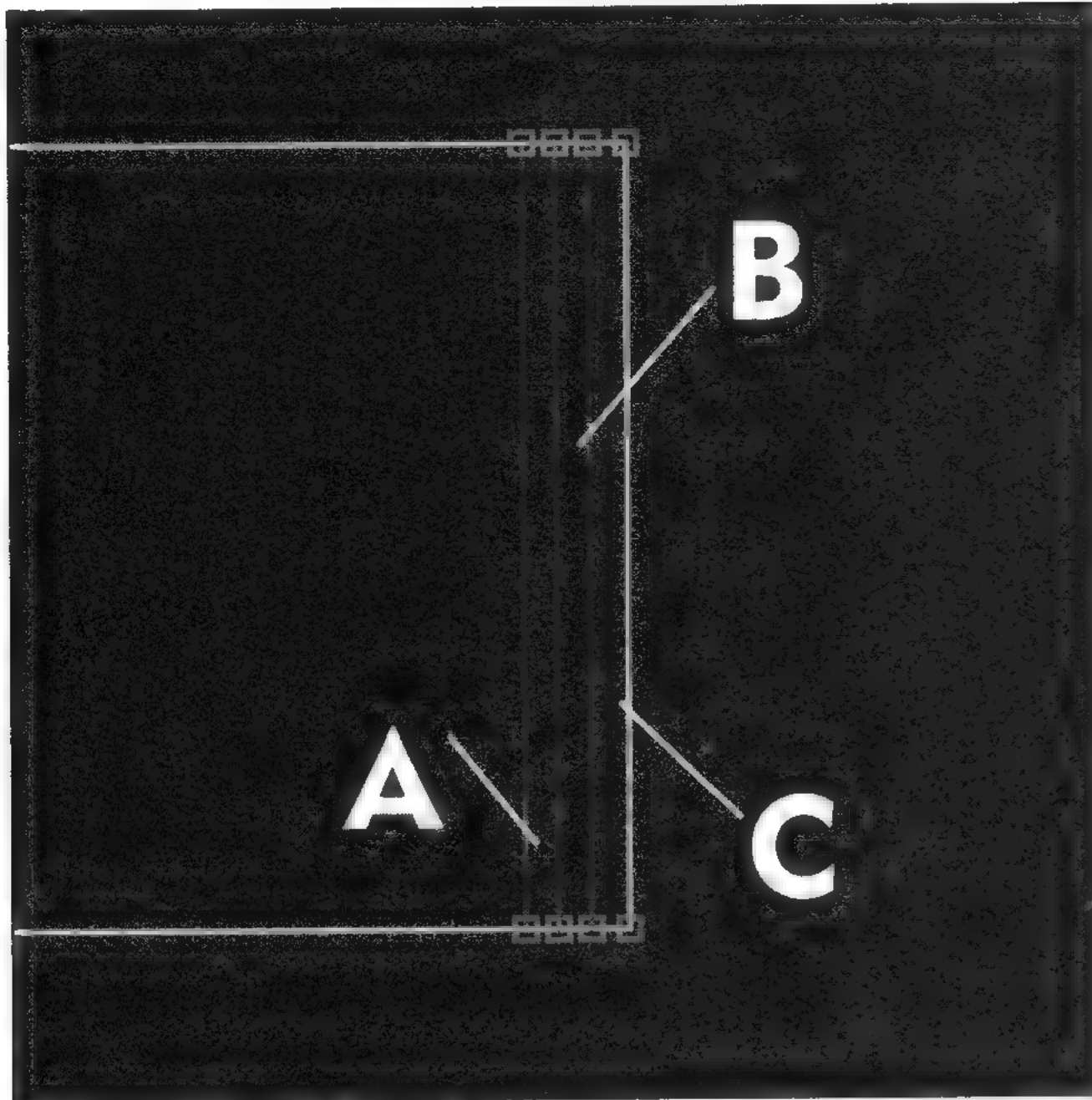


FIGURE 8.15: Set up three parallel sectors at the end of the area that is to have the low perimeter wall.

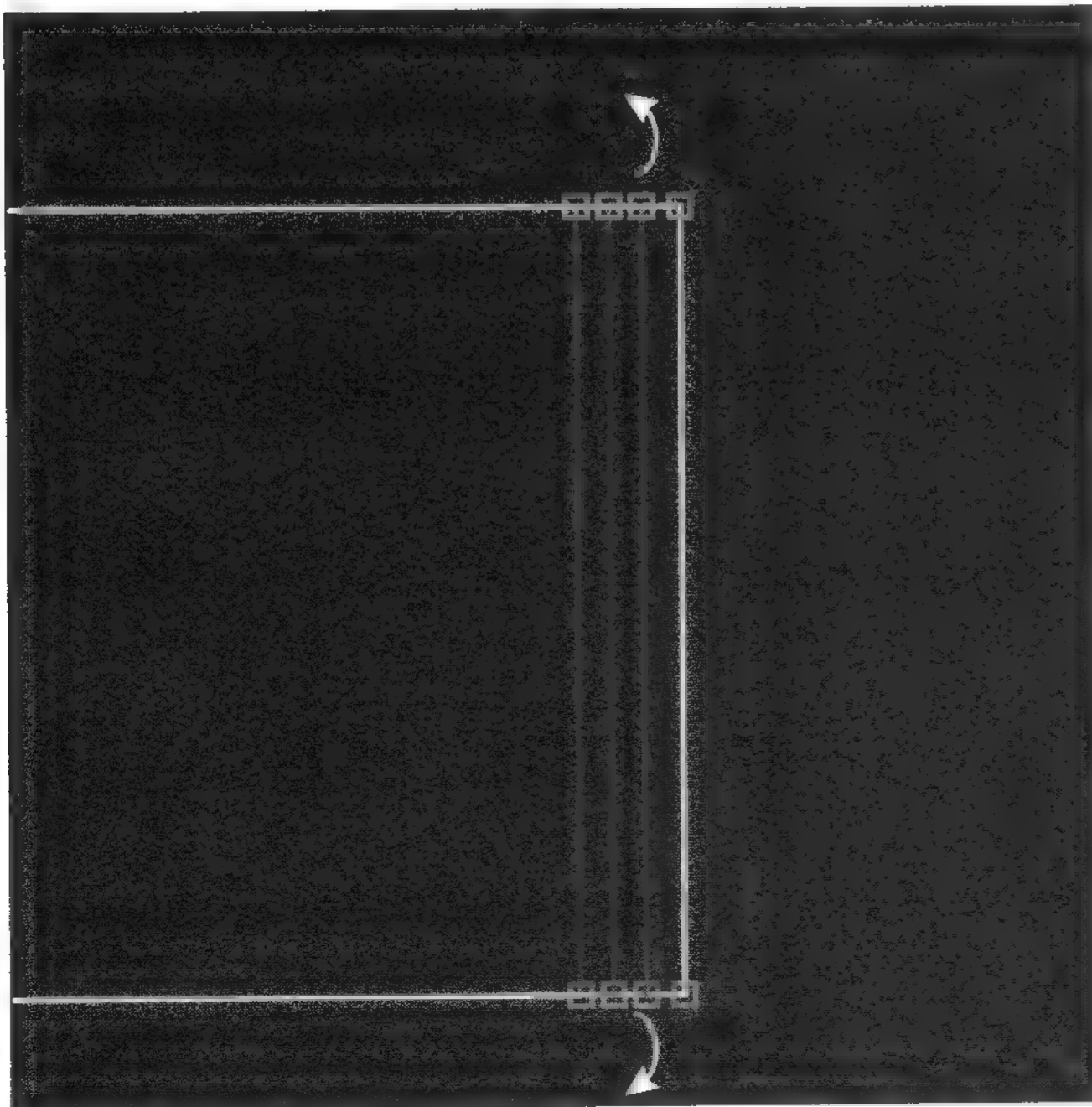


FIGURE 8.16: Move these two vertices to complete the perimeter wall effect.

Sector C, which is the outermost sector, should be given both a parallaxed ceiling and floor. Once this is done, bring the ceiling of sector C all the way down to the floor. Then, select the transparent wall that separates sector A from sector B, and turn on its HitScan and Blocking bits. This will prevent the player and other objects from moving past the low perimeter wall.

Finally, move the two vertices that define the wall between sectors B and C so that they are on the same line with the wall between sectors A and B, as shown in Figure 8.16. This will obscure the side walls of sector B from view.

The low perimeter wall is complete. You now have a natural-looking wall on the border of your level. The final product should look similar to the one shown in Figure 8.17 in 2D view mode.

You can easily turn the low perimeter wall into a window by removing the parallaxing effect from the ceiling of sector A and lowering it until the sector looks like a window, looking out into the open sky. This is a great technique for windows on the edge of your level. You give the player a place to look out at the edge of a level instead of just *ending* a level at the map's edge.

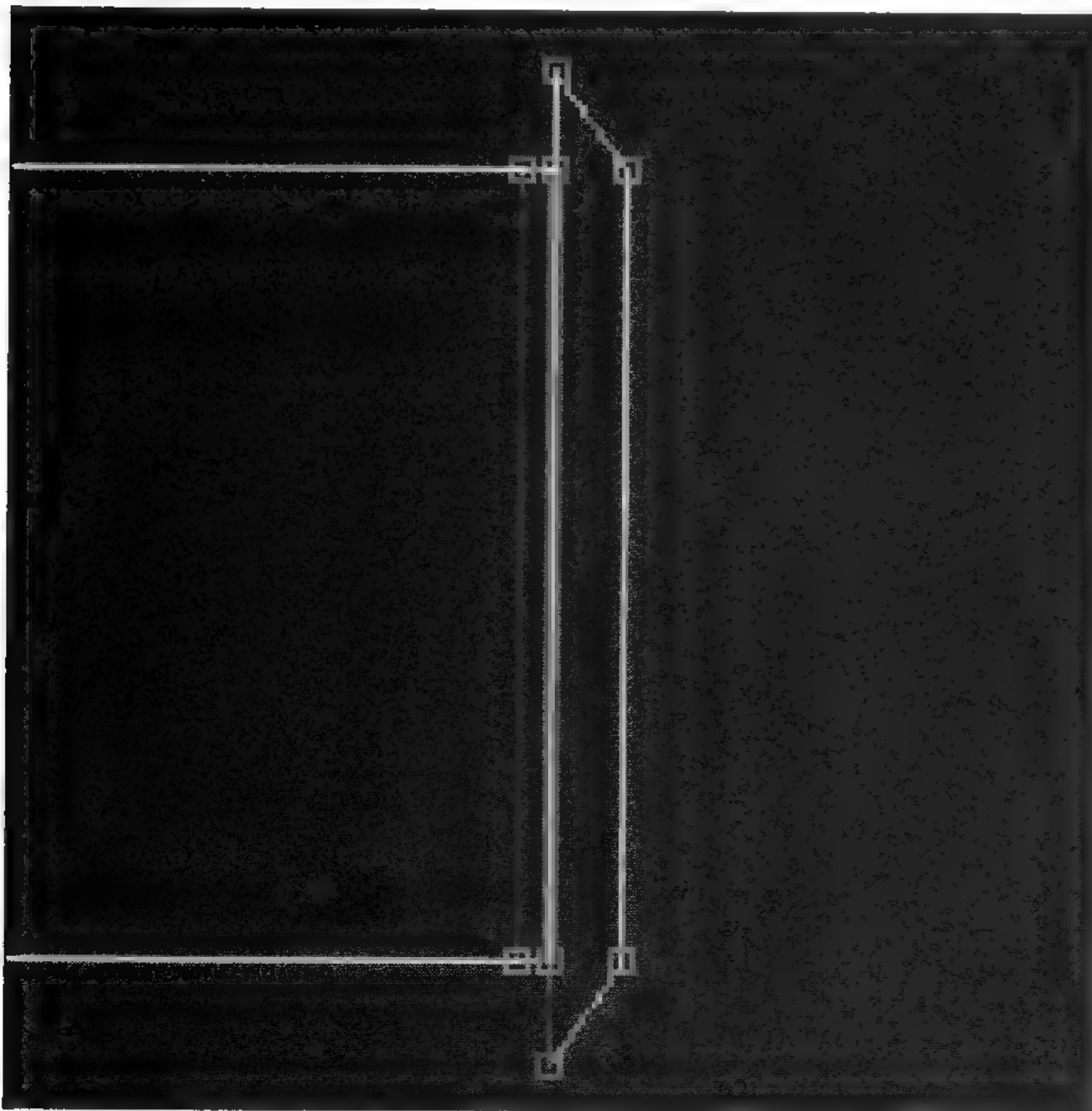
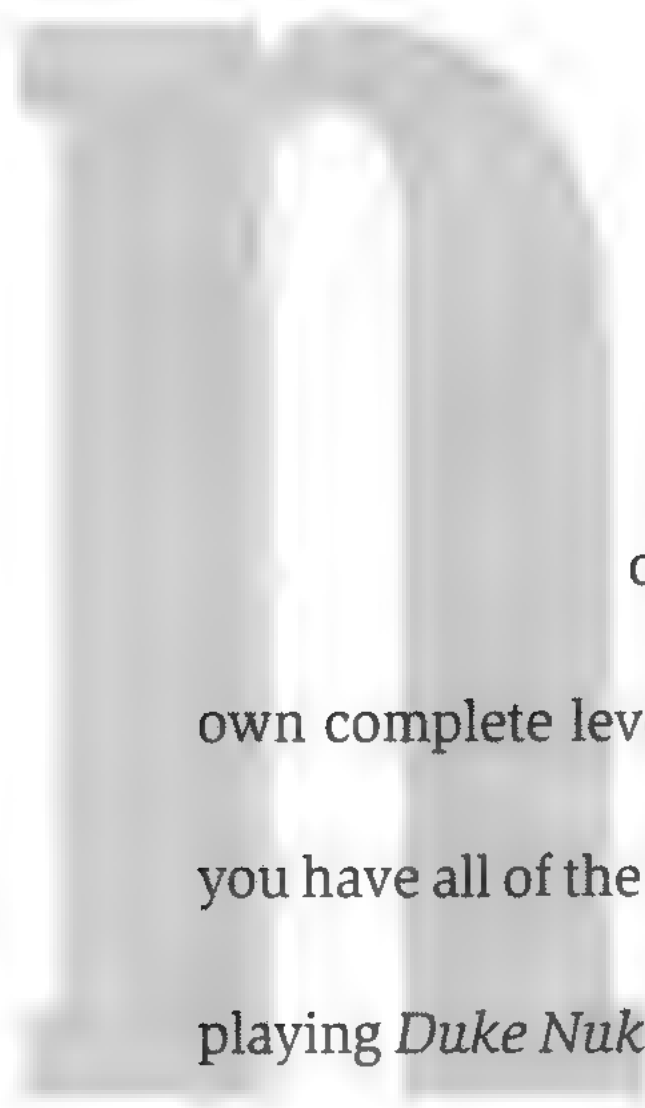


FIGURE 8.17: This is the final layout of the low perimeter wall sectors.





Designing Good Levels



Now it's time to put what you know to the test and begin creating your own complete levels. If you've worked through the previous chapters to this point, you have all of the mechanics under your belt to create the kind of level that will make playing *Duke Nukem 3D* the enjoyable experience you know it can be.

DIFFERENT LEVELS FOR DIFFERENT GAMES

Your first consideration, however, is to decide for which kind of game your level will be best suited. Do you want to create an awesome single-player level, chock-full of all the monsters, weapons, and power-ups you can throw at a player? Or, do you want to create a level that accommodates cooperative play against the alien forces? Perhaps you want to design a DukeMatch level strictly for duking it out with other Dukes. Each type of game requires levels with substantially different design characteristics.

Most level designers believe that making a good single-player level is much harder than making a good multiplayer level. Single-player levels are usually larger and have more puzzles, traps, and monsters than multiplayer levels. In addition, the level designer is responsible for providing *all* the entertainment to the player. In multiplayer modes, the interaction between players (be it friendly or hostile) creates much of the entertainment value. Sometimes a simple, large rectangular room stocked with weapons and ammo is enough to keep DukeMatchers happy, but the single player will scoff at such a level and delete it from his hard drive immediately. This chapter will discuss designing levels for single-player environments; chapter 10 will discuss design considerations for the multiplayer levels.

THE SINGLE-PLAYER DESIGN

So then, what goes into creating an effective single-player level? Since there are so many elements that go into a good level, you'll need a good plan before you go in. Firing up Build and creating rooms haphazardly without any good overall idea or goal in mind will result in an unorganized final product. The best analogy I can come up with are the *real* architects of the world, who spend countless hours in the planning and design phase before anyone ever sets out to put hammer to nail. You won't have to do quite as much planning as these architects, but you will have to do some work up front to create your final goal: the level that makes everyone say "wow" after they play it. Design for a single-player level involves the following general phases:

- ❖ Design on paper
- ❖ Do initial layout
- ❖ Place major items
- ❖ Complete structural aspects
- ❖ Place monsters
- ❖ Place weapons and ammo
- ❖ Place power-ups
- ❖ Create scenery
- ❖ Do final testing

Designing on Paper

I know that it's every level designer's urge to rush to the computer and start construction right away. I can almost guarantee you that starting off in this manner will result in a level that nobody will want to play. You need good, solid ideas for your level before you begin. The first tools you use in your quest to create a new level should be a paper and pencil. Write down the ideas you have so far, either as sentences or as rough sketches of your map.

As you rough out these ideas, try coming up with an overall *theme* for your level. Will your level be on a city street or in the void of space? Try to fit your theme into the overriding theme behind the entire *Duke Nukem 3D* story, which of course is the invasion of

Earth by alien forces. To what extent has this invasion taken place in *your* world? Are the aliens just arriving, or have they already established a strong foothold? The answers to these kinds of questions will help you later in deciding textures, monsters, and lighting.

On paper, try to break up your idea into discrete sections that will in turn translate to the buildings, rooms, or other sectors of your map. During this process, pay attention to the *flow* of the level. Determine where the player starts and ends the level, even if they're just general areas like *north* or *in a small entry room* for now. The most obvious place for the end of the level is on the opposite side from the level's starting point, but it's also possible to put the end very close to the starting point, even in view of the starting point, but unattainable because of a locked door or force field.

This is also the time to decide the approximate difficulty level of your map. Placing a boss around every corner is too difficult, obviously, but placing nothing but assault troops in every area might be *too* easy. If your level is to be part of a series of levels, or even an entire episode, the placement of this level within that series will also help to determine the approximate difficulty of this map. If it is to be the last map in an episode, you'll want an appropriate area for the player to tangle with one of the boss monsters. If the level is to be early in the episode, then there shouldn't be any of the tougher creatures. For mid-episode maps, a nice mix of different creatures is often appropriate.

In these single-player levels the player will start only with the default weapon and no power-ups; the player won't inherit any items from previous levels. Therefore, you might want to get the player a shotgun or a ripper chaingun early in the level. Alternatively, you might want to design a level that challenges the player to survive with only the default weapon for a while. Start thinking of these considerations early, as they will give your level shape and its own *personality*.

On paper your level should start to develop a shape. Give the rooms' shapes *variety*. This variety can be nonrectangular rooms, varying ceiling and floor heights, sloped floors and ceilings, and special features like water, subways, and elevators. Make sure to use the Z plane to your advantage. *Duke Nukem 3D* levels can have sectors over other sectors (to a certain extent), and this feature further enhances the realism of the level.

Finally, make sure that each area has a logical purpose for being in your level. That is, make sure that the room contributes to your overall theme. Don't put a munitions bin in the middle of a hospital, because there's no logical reason such a thing would be present there. However, a large weapon cache inside an apartment is much more believable. Even though you may be creating fictional areas like alien spaceships, your levels should be *believable* fictional areas.

Doing Initial Layout

Once you have a plan on paper, you can start the building process. Fire up Build, and start outlining the general shape of the major sectors. You don't necessarily have to start at the beginning of your level in this phase. If your level features an important central area, you may want to start in this location to make sure there's room for it on your map.

Pay close attention to the height of each sector and the way that each major area is going to connect as you rough out your level's major rooms. Try to create areas that you can see but not reach right away. This will give the player a good sense of direction and the clues necessary to solve your level. A simple example of this is a locked door with a window nearby that the player can peer into.

You can start picking out textures for your sectors at this stage, but don't spend too much time getting every texture perfect just yet. As your level takes shape, you're bound to make fairly significant changes, and giving an area the perfect textures, shading, monsters, and so forth may be a waste of time. You will likely end up hacking out half of the initial area to accommodate a better idea. Just get the overall shape of the level down at this point.

During initial layout you can also start creating the special sectors like lifts, doors, teleporters, water, switches, and all the other effects that contribute to the flow of the level and allow the player to get from place to place. Of course, you'll have to test these effects in the game because they don't work in Build itself.

Once the initial layout of your level is down, you should be able to walk from the intended start to the intended end of your level. Use the cheat DNSTUFF when play-testing at this point to give yourself the keys, jetpacks, and so on necessary to complete the level's walk-through.

Placing Major Items

The next phase of design is to find places for the most important items in the level. These include the player start locations, the locked door access slots, and the keycards to open these doors. Also place any jetpacks necessary to complete the level. Then, play-test the level again, this time *without* using any cheat codes, to make sure you don't inadvertently place a keycard in an inaccessible area (like putting the red keycard behind a door that requires a red keycard to open it).

While testing during this phase, you're bound to go back and add some new areas or adjust some existing ones. By the end of this phase, however, you should be able to run through the level completely without any cheating. Once this has been accomplished,

you've successfully created the *flow* or *critical path* of your new level and can now move on to the next step.

Completing Structural Aspects

This phase of design is where you will give your level its *mood*. Your goal here is to complete the structural part of the level. This includes selecting textures, adding shading to each area, and adding any additional doors, mirrors, glass walls, force fields, or any other structural feature. The overall layout of the level shouldn't change much anymore, but you will still probably make small additions and subtractions as you go from area to area.

The best methodology to use when working on this phase is a room-by-room methodology. Start with any area and work on it until you think it looks perfect. You will probably add quite a few new sectors during this phase to create shadows in what used to be large single rooms. Keep in mind how important light and shadow are for adding realism to the level. One way to do this is to ask the question, "Where is the light coming from?" in each and every sector of your level. Then design the level with those light sources in mind. If the light source is a window, then cast shadows on walls opposite that window. If there are overhead lights, then shade the room accordingly.

Don't forget to use special effects like allowing the player to shoot out the lights to make a room go dark or having areas light up when a door opens. If it makes sense to do so, put light switches in your levels to brighten up dark areas.

Texture selection is also critical during this phase of development. Choose textures to match the given area, obviously. Don't try to use every texture available. Less is often more when choosing a group of textures for your level. Keep in mind that the same texture can be made to look different by scaling, panning, or shading that texture.

Also recall that areas that are initially unconnected can become connected later with the use of exploding holes or earthquakes. These types of paths can open up areas that a player could see but could not get to in earlier parts of the level, or they can connect a new area to an older one, if that area suddenly gained new importance (for example, a force field turned off to allow access to a new hallway).

Much of this phase of design can be accomplished without having to go into *Duke Nukem 3D* itself, which is why Build is such a powerful level editor. You'll be able to see almost exactly what your level will look like in the game, with the exception of the moving sectors. Even though you won't waste time hopping in and out of Build, however, by no means should this phase of the design be done quickly. As mentioned many times earlier, your level will not be any good if it isn't believable, and appropriate textures and

realistic lighting are two of the most important design elements that can make the level believable.

Placing Monsters

Now comes the time to lay out the different combat scenarios for your level. Many of these scenarios probably made themselves known to you during earlier phases of the level's creation. The best approach to this part of the level's design is a room-by-room approach. Place each group of monsters in each major area, one area at a time. Here it will be necessary to go into the game and thoroughly test each combat scenario. Use the DNSTUFF cheat while testing to give yourself all the weapons you'll need, but don't just waltz through your level pulverizing everything with the devastator. Try to use the weapons the player will have when going through the level. You'll also want to use the DNKROZ (God mode) cheat at this point so you don't die while testing each room.

You'll also want to keep in mind the different skill levels within *Duke Nukem 3D*, and add more or fewer monsters to each skill level to make the level more attractive to different kinds of players. You may be an expert player, but remember that not everyone is, and your level should be made easier for these types of players. Also remember that no matter how awesome a player you are, someone out there is probably better, and you'll want to add monsters that make the level harder than you can accomplish without cheating at the hardest skill level.

During this phase, continue to make minor adjustments to the structure of your level, if necessary. For instance, you might want to create new hiding places for some of the monsters.

This is also the first phase, of level design where frame rate might come into play and alter your ideas. If your level gets too complicated in any one area, the game will slow down considerably, and your level simply won't be fun. If a decision has to be made between the *coolness* of an area and frame rate, the latter should win out every time. Poor frame rate will kill off an idea no matter how cool it is. Also keep in mind that people may be playing on slower machines than your own, and you shouldn't keep adding stuff to your level until you see the first hint of a slowdown. Those poor saps on 486-33s that can just barely get *Duke Nukem 3D* running will never be able to play such a level.

You also want to take into account the type of monster you use in each area and how the monster might use an area to its advantage.

Placing Weapons and Ammo

In this phase of design you'll decide how you're going to allow the player to obtain arms. If this level is to be one of a series, weapon placement becomes even more important. (You don't want to give out a devastator two minutes into the first level of an episode.) If the level is a true stand-alone level, however, many designers like to give a nice array of all the different weapons and ammo.

First decide exactly which weapons you will and won't give out. Then, find a suitable location for each of these weapons. Recall that certain enemies will give out a weapon when they perish, so take this into account as well. The more powerful weapons should be challenging to acquire; they are often used as rewards for completing a difficult part of the level or solving a puzzle.

Once the weapons are placed, you need to place the ammo for those weapons. Ammo placement can be difficult, however; placing too much can make a level too easy, while placing too little can make the level too hard. Determining how much ammo to use is best done through play-testing. Enable the God mode cheat (DNKROZ) when you play-test, and give yourself all the weapons and ammo (DNSTUFF). This time, however, only use a weapon once you've come across it in the level. As you go into each room, dispatch all the creatures there, then write down your ammo levels for each weapon. Then, type DNSTUFF to bring all your ammo levels to maximum again. Repeat this for each area in your level, and you will determine how much ammo is needed to complete each major area.

Then, go back into Build and distribute the ammo. It often makes the level more challenging to place a bit less ammo than was required during your play-testing, which will force the player to become more creative. For example, the player may have to use the laser trip bombs a bit more than usual or have to dispatch some monsters using only a pistol. Don't make things *too* hard on the player, though; your level won't be fun to play if it's impossible for the player to finish it.

Finally, play-test your level one more time, but this time do *not* give yourself all the weapons and ammo. Make your only source of weapons and ammo the level itself. (You can leave the God mode cheat on, since there isn't any health in the level yet.) You should be able to get from the start to the end of your level.

Placing Power-Ups

Placing the appropriate amount of atomic health units can be done the same way you place ammunition. Go through your level in the game, and take note of your health levels. Then, go back into Build and place atomic health units judiciously throughout your map. Keep in mind that the player doesn't have to be at 100 percent health or above throughout the entire level. One of the things that makes a level more challenging is having to face a combat situation with very little health available. This makes even the lowly assault troopers more challenging, as even a single shot from them can mean the end of the player. When placing atomic health units, keep things balanced so your level does not become too hard or too easy.

Don't neglect the healing benefits of water sources like drinking fountains or broken toilets. Don't let the player load up on health for free; make the player earn a drink of water by clearing out a crowded room first. As for other power-ups, some of them may have already been placed if they were necessary to solve the level (like jetpacks). Now comes the time to place any other items that will help the player. Find a good spot for the scuba gear if your level has water; include night vision goggles to get through any dark areas in the level (especially if you hide a message on the wall in one of these areas); and spread steroids, boots, and holodukes as you see fit.

Creating Scenery

The last phase of level design shouldn't affect the overall play of the level much, but it can do a great deal toward establishing that always-important mood. Place some scenery sprites around the level for added realism. Items like wall calendars, bottles, and signs on the walls can make your level look like a real place. This process works best as the final design phase because some of the more complex areas might suffer in frame rate with the addition of too many scenery sprites. You can always leave them out of the more complex areas, because these items are the most expendable.



TIP

If a once normally functioning area suddenly suffers from a slow frame rate after you add scenery sprites, this may tip you off that you went a bit overboard with monsters in the area. You might want to tone down the area so that you can place a few scenery sprites around.

Unfortunately, this step is not the most fun part of designing the level. Try *not* to skip the step entirely, as those added little visual items may just be the thing that turns your level from merely good to great—one that players will play over and over again.

Doing Final Testing

By the time you reach this phase, you should be completely tired of your level and ready to start a new one. But there's still a bit more testing to be done. Go through the level again, trying to imagine that you're a player seeing it for the first time. Try not to anticipate creatures that you know are around the corner, and see how the level *feels*. Then, if possible, try to get someone else to play it, and watch the person as he or she plays. See if your carefully designed traps work. Ask the player for feedback. From things you might observe and some constructive criticism, you can go back and make those extra one or two adjustments.

When you feel that your level is flawless, fill out a MAP Authoring Template (one is included in the Build documentation), and upload your level to your favorite online sites. Make sure to include your e-mail address and to ask for any constructive criticism or comments about your level. Finally, your masterpiece is complete!

GOOD LEVEL DESIGN CONSIDERATIONS

What follows are a few more tips for creating a decent level. These are things that may seem obvious at first glance. However, many of these little techniques might easily be forgotten among the myriad little details that need to be considered when designing your level. Make sure to run down this checklist after your initial design is complete, to see if you capture at least some of these elements.

Adding Suspense

Sometimes, a steady stream of enemies for the player to fight can get monotonous. Think of how many times you've heard the old cliché, "It's quiet...*too* quiet," in a movie. Leaving areas devoid of monsters can serve a few purposes: It can add suspense to the level, and it can give players time to catch their breath after a nasty battle. Sometimes you can leave an area free of monsters just to give the player a chance to admire a particular area of your level that you're proud of. (A player has a hard time enjoying the

view when being shot at.) Consider also that because the game is technically a shoot-'em-up, having an area with no monsters just doesn't *feel* right sometimes, which can make a player a bit uneasy. In the level designing business, uneasiness is good!

Surprise, Surprise, Surprise!

Ambush and surprise tactics are a potent weapon in the level designer's arsenal. There are many ways to induce surprise in a player. Most of them involve placing a monster or explosive device where the player won't be expecting it. Here is a brief list of good ways to surprise the player:

- ❖ Use the Respawn sprite to teleport creatures when a TouchPlate sprite is stepped on or a Switch sprite is activated. This can cause a previously cleared room to suddenly repopulate in a hurry.
- ❖ Use wall-mounted laser trip bombs that activate when a panel is raised behind a passing player. They make a great means of blocking an escape route.
- ❖ Use darkness to your advantage. Put small alcoves in shadow and place monsters in them. Or, use large darkened areas to promote confusion and give your monsters lots of different places from which they can pop out. Something as simple as a dark room and two monsters suddenly appearing on opposite sides of a player can be devastating.
- ❖ Use water as a place to place surprises, because the player basically *pops* either into or out of the water into a new sector. How many devious little things can you think of to place in an innocuous-looking pool?

Establishing Difficulty Levels

Keep in mind that not all players are as good (or bad?) as you might be. *Duke Nukem 3D* attracts people from their early teens all the way into their thirties and beyond. With these different age groups will come different skill levels. The good level will cater to players of all skill levels, from the young guy playing a harmless game with

his buddy, to the grizzled Duke veteran who just wants to shoot up a few levels before going to bed to rest up for that 9-to-5 job the next day.

The best way to create different difficulty levels is to start by designing the level to match your own skills. Then, take a few of the monsters you've already placed and mark them to be at a higher skill level, to make the game easier for the novice players. This is especially easy to do in areas where groups of monsters exist. Then, add a few *more* monsters to key areas, and mark them at the highest skill level, a level you would have great difficulty beating yourself. To set difficulty settings, give a sprite a LoTag as follows: Lotag 0 for all difficulty settings, Lotag 2 for "Let's Rock" and harder. Lotag 3 for "Come Get Some" and harder settings, and Lotag 4 for "Damn, I'm Good."

Don't forget that adding and subtracting monsters is not the only way to make a level harder or easier. More or fewer atomic health units, ammunition, and power-ups can serve an equal purpose. You can also take away the more powerful weapons from the expert players.

Building in Risk and Reward

This simple maxim basically states that if the player completes a particularly tough section of your level, you should reward the player with some type of power-up, key-card, or other reward. Put the opposite way, if you place a particularly powerful or necessary item in your map somewhere, don't just let the player waltz in and take it! Make the player *earn* it.

Offering Puzzles and Solutions

There are many ways to create nearly impossible puzzles or traps that will pretty much instantly kill a player. A hidden teleport into a cramped area with five Battlelord Juniors, for example, is one quick way to get the player killed.

These types of traps are also a quick way to get the player to leave *Duke Nukem 3D* and delete your carefully made level from the player's hard drive. Keep in mind the age-old law of computer games: *It's fun to win them*. Sure, you want your level to be *hard*, but if it's instead *impossible*, many players won't like it. If you do have a penchant for creating really devious traps, there is an audience out there for levels of this type. Just make sure to advertise this level as a really nasty one in your MAP authoring template so that your level will get into the hands of those who can appreciate it.

For most types of puzzles, there should be a fairly easy solution or at least some type of hint to help in the solution. Messages on walls in dark areas was a common method of giving hints to the player in the original levels. If you choose this type of hint, make sure to put night vision goggles somewhere on your level so the player is sure to have the chance to see your hint. Other types of hints might be arrows on the wall or a slight shade change that hints at a secret area behind a wall.

Just keep in mind that your level needs to be ultimately solvable to be enjoyable. Be aware of the fine line between challenging and impossible.





Designing Co-Op
and DukeMatch Levels

here are two ways in which *Duke Nukem 3D* can accommodate multiplayer games: cooperative (Co-Op) mode and DukeMatch. In Co-Op mode, a group of players teams up to thwart the monsters and get through the game's levels. DukeMatch, on the other hand, allows you to play against other Dukes (human opponents). In this chapter, you will see how level design for these types of play modes can be somewhat different from designing single-player levels. The first section discusses level design for cooperative play, and the other addresses design considerations for DukeMatch levels.

DESIGNING LEVELS FOR COOPERATIVE MODE

Most levels designed for single players also work well for Co-Op mode; the design goals of the level are in many ways the same. The level designer is required to supply a critical path for the players to discover and complete from start to finish. However, here are some design considerations for a level that's specifically suited to Co-Op play.

DESIGN OPEN SPACES

A level specifically suited to cooperative play should have slightly larger areas to accommodate the extra players. Even simple hallways should be made wider so that players can stand abreast and handle the obstacles together. Of course, certain areas can be intentionally left more narrow to constrict combat. However, much of the fun of a Co-Op level is the ability to battle simultaneously.

INCLUDE MORE MONSTERS

The more players there are in your level, the more lead will be flying around. Therefore, it's only fair that you supply more targets for the players. Feel free to add more monsters to make the battles more difficult. Be sure to pay attention to your level's frame rate. Don't indiscriminately add monster after monster until your level slows to a crawl. Keeping the monsters coming at a steady pace a few at a time instead of large groups all at once is one way to add enough enemies without decreasing frame rate.

ADD MORE OF EVERYTHING!

Any item that a player can acquire will have to be in greater supply in a Co-Op level. Weapons placed in the level reappear after a brief time in multiplayer modes, so the game handles this for you automatically. As for health, you can either simply add two or more of everything right next to each other, giving healing benefits to all players at the same time, or you can stagger the supply, which allows the players to decide who needs each atomic health unit or medkit the most as they happen upon it. Also remember the healing available in the form of water sources like toilets and drinking fountains, which can be added once to supply healing for everyone, without any risk of reducing frame rate because of overloading on sprites. Ammo should be more abundant, as well, to keep everyone trigger-happy throughout the level.

DESIGN FOR FRIENDLY FIRE EFFECTS

Creative level design can make for situations in which a player might attempt to unload ammunition on a nearby enemy and instead damage a fellow teammate! Having monsters come from multiple directions is one way to accomplish this, and so is the intentional design of more constricted places that make it harder for the players to shoot around each other. Keep in mind, however, that an entire level designed in this way looks more like a designer didn't take the time to convert a single-player level into an effective Co-Op level.

INCLUDE EXPLOSIVES THAT MAKE YOU GO BOOM!

Relating directly to the subject of friendly fire, exploding items are great in Co-Op levels. C-9 canisters and fire extinguishers can easily become deathtraps for trigger-happy Co-Op players. Try to find interesting ways to disguise these traps, so players may not see them right away when starting a major conflict. One example would be to place

C-9 canisters on either side of an opening door, and put monsters directly behind the explosive devices. Players running in the door, turning, and firing immediately are in for a surprise if a C-9 canister is suddenly between them and a monster!

DEVELOP COOPERATIVE PUZZLES

Certain puzzles can be made solvable only in Co-Op mode. If you place a switch and the door it opens far apart, a single player may not have time to flip the switch and then run to the door before it closes; however, a second player can simply stay by the door while the first player flips the switch. Such puzzles should be used with care, however. If your level does have puzzles that can only be solved by more than one player, you should advertise this heavily in your MAP authoring template, or a single player could become frustrated trying to solve a puzzle that's impossible for a single player to solve. You also have the option of making the switch for this kind of puzzle a multiplayer-only switch by setting the palette to 1, which will make it invisible in single-player mode.

SET UP DIVIDE AND CONQUER SITUATIONS

If you can split up a group of players, you can provide additional challenges, especially if each area is packed with extra monsters designed for cooperative play. There are many ways to split up the players. One idea that immediately comes to mind is a rotating sector with two teleporter pads on it, with each pad leading to a different area of the map. The rotation of the parent sector may be enough to fool the players into hopping on different teleporters. Another idea is a simple door that tries to separate players by closing quickly, especially if that door requires a keycard, and only the first player has the key!

Another way you might quickly split up Co-Op players, if even for a brief time, is a simple above water-underwater sector pair. The water surface blocks sight lines from sector to sector, and a player above the water won't be able to see what's going on under the surface unless he or she jumps in.

USE THE SHRINK RAY

The shrink ray can create a particularly nasty trap in cooperative play modes. Since players automatically crush actors shrunk by a shrink ray, you can try to create a situation where one player gets shrunk while standing close to a teammate. If you can successfully create this situation, you'll be able to take out a player with one shot.

Another little-known fact about the shrink ray is that a mirror will reflect it back to the player! An imaginative and devious level designer might be able to get a player

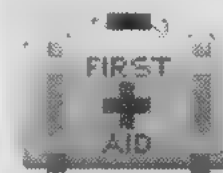
to shrink himself. If his teammate is within stomping distance, then the player just committed suicide, since the stomp is automatic if a player is within range of *any* shrunken object.

DESIGN FOR UNFORESEEN SITUATIONS

Many times when designing a single-player level, it's easy to account for the whereabouts of the player, which makes it easy to design a trap for the player. For example, because a touchplate requires the player to walk into a sector to trigger an action, it's easy to anticipate exactly where the player will be so that you can spring the trap.

These same traps often fail to work as planned in cooperative play, however. Even if you get a player to walk over an intended sector to spring a trap, the X factor becomes the location of the second (or third, or fourth) player. That extra player's location might serve to defuse the trap. For example, if a trap is designed to open a door behind a player standing on a touchplate, a trailing player might see the door open and alert the first player to turn around.

Because there are so many variables to consider, don't try to devise the perfect cooperative mode trap. You may just end up wasting all the extra work. Instead, concentrate on simpler traps like monster ambushes, explosive traps, or simple dark areas that tend to disadvantage all players equally.



TIP

It's easy to slap the word **cooperative** on your **MAP** authoring template for a level you designed as a single-player level. Taking a few simple extra measures, however, can turn that great single-player level into an equally good cooperative-play level. Many times, these extra steps won't detract from the playability of the level in single-player mode, which means that your level will appeal to a wider audience.

CREATING MEMORABLE DUKEMATCH LEVELS

Oh, the joys of blowing your best friend into small piles of virtual goo! For some, DukeMatch play is the *only* way to play *Duke Nukem 3D*. What DukeMatch offers that single-player levels don't, of course, is the chance to match wits with human opponents, who are trying just as hard to figure out how to blow you to kingdom come.

There are different issues to consider when designing a good DukeMatch level. All of the visual design issues discussed earlier still apply, of course, such as the effective use of texture and lighting and having a consistent theme for the areas in your level. The additional considerations that follow may well be the difference between whether people all over the world are slaughtering each other in levels of your own devious design, or if they try your level out for five minutes and then consign it to their MAPS directory with 14,000 other downloaded levels.

BALANCE, BALANCE, BALANCE

These are the three most important things to consider when designing a good DukeMatch level. If each player doesn't have an equal chance to win, then that player won't play your level again. If one player won't play your level, then of course his or her DukeMatch partner(s) won't be playing it either, which means your level is a failure. To create balance, keep the following considerations in mind:

- ❖ Place player starting locations an equal distance away from weapons.
- ❖ Divide health, ammo, and other power-ups equally around the map.
- ❖ Allow multiple means of escape from each starting location. This way, an opponent can't *camp out* near the only means of exit from a starting location to ambush another player there.
- ❖ Make weapons of equal power available equally to all players; don't let one player have immediate access to a case of pipe bombs while another only has a shotgun.
- ❖ Include fewer secret areas containing extra power-ups. This creates an unfair advantage to anyone who knows a level over an opponent who's never played it before. Weapons and power-ups should be out in the open or at least not too terribly concealed.

SHRINK THE LEVEL

Bigger is not always better when designing a DukeMatch level. If the players spend most of their time trying to seek each other out, they will get bored quickly. This is why DukeMatch levels are often smaller than single-player levels. Make sure you can

get from one far corner of your DukeMatch level to the other rather quickly. Ten seconds might be a good maximum time for such a trip.

CREATE A NONLINEAR DESIGN

Many *Duke Nukem 3D* levels designed for single-player mode don't make good DukeMatch levels because of all the attention paid to the flow of the level. Most single-player levels have what Levelord, one of the principal designers of *Duke Nukem 3D*, called a *critical path* that needs to be followed in order to get from the start to the finish, and it's the player's job to find that critical path. In DukeMatch levels, however, there's no need for a critical path because the goal of the level is *not* to find the end, it's to turn your opponent into pulp! Therefore, the design of DukeMatch levels is often much less linear than single-player levels; that is, many areas have multiple ways of entering and exiting them, and each exit can branch off into many other directions. Try to design your levels so that there are at least two ways to get from any point A to point B on the map. This way, a player at point A doesn't know which direction his opponent is going to come from, even if this player knows the opponent's approximate location.

CREATE LARGE COMMON AREAS WITH MANY ENTRY AND EXIT POINTS

Arena-style levels often make the best DukeMatch levels. A large common area, or arena, lies in the center of the level, and many passages lead out of the area radially and interconnected outside the arena. Much of the fighting can take place in the arena, and a mortally wounded player can hightail it out using one of the many exits to locate health or ammo.

These types of DukeMatch levels are very common, so if you do decide on this central idea, try to think of something to set your level apart from the others. Perhaps the arena is much higher or lower than the rest of the levels. Perhaps it is rotating, like the DukeMatch level *Spin Cycle*, which is included with the retail version of the game. Perhaps there's a boss wandering around the arena providing an extra obstacle. Perhaps walls can be made to raise and lower within the arena, providing additional cover. Because these types of levels are so common, yours is going to have to be different. Put on your thinking cap a bit longer for the standard arena level.

CUT DOWN ON THE PUZZLES

Keep the puzzles to a minimum when designing your DukeMatch level. Most players don't play DukeMatch to challenge their problem-solving skills—they play DukeMatch to reduce their opponent to a damp spot on the floor!

That being said, a *few* well-placed puzzles can enhance the playability of the level, as long as the puzzles aren't extremely difficult to figure out. Perhaps finding a certain hidden switch can reward the player with one of the more powerful weapons on the other side of a locked door.

Another thing to keep in mind is that good DukeMatch levels are often played over and over by the same group of people. If this is the case with your level, any secrets or puzzles will have a fleeting impact because by the second or third time a group of players go through your level, they will know all the secrets and solutions to the puzzles by heart.

DITCH THE MONSTERS

Monsters usually detract from the point of a DukeMatch level, so many designers don't even bother putting them in. However, a DukeMatch level that contains a single Boss wandering around provides an excellent obstacle or distraction. For example, consider placing a Boss on a raised platform in the center of an arena. Maybe the monster is usually trapped and can't see the players, and a switch is used to raise a door to allow the monster to fire. This creative use of a monster makes an excellent trap; for example, a player can hide out by the switch and raise the door when an opponent appears. (In this case, make sure the area near the switch is hidden from the monster's view, or the player flipping the switch would be committing suicide!)

Also remember that players can easily disable the monsters in your level by using the /M code on the command line when playing the game, so don't make the monsters the only interesting point in your DukeMatch level. You'll have no control over the players turning them off.

ESTABLISH AMBUSH POINTS

Players need a place to hide from each other when playing DukeMatch levels, so make sure to design plenty of hiding places. Consider some of the following ideas for ambush points:

- ❖ **Water areas:** Remember a player can fire weapons into and out of water sectors, so they make excellent hiding places. Place scuba gear nearby so that the player can stay hidden a long time.
- ❖ **Dark sectors:** The best cover is darkness. Make sure some of your sectors are dark or have realistic shadow effects.

- ❖ **Sniper points:** Small raised areas overlooking larger areas make good cover. Make sure there's a wall the player can duck behind for additional cover.
- ❖ **Murder Holes:** These are narrow windows just wide enough to launch a rocket through, located along one side of a hallway. They provide great cover.
- ❖ **Elevators:** The elevator to the arcade on the Hollywood Holocaust level (E1L1) is a good example. A player can wait at the top, and when an opponent summons the elevator, the offensive player can drop a pipe bomb or two in it and wait for the player to step on.
- ❖ **Bridges:** Although Build doesn't support true bridges, a sprite-based bridge can provide a cool hiding place for an overhead ambush. If you design the bridge so that it can be destroyed, then this adds yet another dimension to the area. If the destructible bridge is over a pool of harmful green slime, then yet another feature is added.
- ❖ **Moving sectors:** Subways, two-way trains, and rotating sectors are unique to the Build engine. They can quickly bring a player from Point A to Point B to create an ambush. Can you imagine staring down a dark tunnel with your enemy bearing down on you, guns blazing, on a subway? Very evil.

USE THE Z PLANE

Keep in mind that your world is a three-dimensional one (well, *almost* a three-dimensional one). Try to think *three dimensionally* when designing your levels. Remember that you can make normally unreachable areas reachable, simply by including a jetpack nearby. With the added benefit of sectors above other sectors and false sprite-based floors, your levels can be designed with players directly over or under one another.

Using the Z plane in your levels also gives you a new force for the players to contend with: gravity. Gravity can help a player dropping a pipe bomb from a ledge. It can also hurt a player floating along with a jetpack that suddenly runs out of fuel!

USE MOVE SECTORS

These are discussed previously in conjunction with ambush points, but there's no end to the types of things that can be done with moving sectors. Exploding C-9 canisters

can rotate into a room, *shelf* sectors can rotate in circles causing the player to wait for them to come around before he or she can grab the goodies thereon, and subways and two-way trains can provide quick escape routes. Even the simple swinging door provides a new means of cover for a player that *DOOM* couldn't provide. (The player can stand to the side of the door and still be under cover when it opens.)

ADD SOUND AND OTHER CUES

Clever use of sound in the level can add to the tension of stalking and being stalked. Make sure your doors and lifts make noise, so players can use them as clues to their opponents' whereabouts. Try to vary the types of sounds you use, so that players can associate a given sound with a certain area of your map.

Other cues can be as simple as leaving on a light or special sectors that point which way a player left a room by illuminating his or her path. Giving your level this kind of depth will make it more playable.

GIVE PLAYERS THE TOOLS THEY NEED TO SET TRAPS

Laying traps for an opponent to stumble into is one of the most satisfying methods of dispatching a DukeMatch opponent. One of the best places to lay a trap is in an opponent's anticipated escape route from some adversary.

Set Wall-Mounted Laser Trip Bombs

I've heard people say that laser trip bombs are useless because everyone can see the red beam. To that I say HA! These people just haven't thought hard enough on how to use them. There are plenty of ways to place a laser trip bomb without the player seeing the laser. Consider some of the following locations for this devious weapon, where its beam can be projected:

- ❖ Low over the surface of a body of water so that it triggers when the player surfaces.
- ❖ In the path of a swinging door.
- ❖ Inside a hidden small door panel so that it explodes when the panel is opened. This is *especially* devious if the panel originally contained an artifact that the player is sure to come looking for.

One final note about laser trip bombs is that the player may not see the laser simply because he or she is running backwards. If a player sets up an ambush and lays a laser trip bomb on an opponent's expected escape route, there's a good chance the opponent will be running backward to cover his or her escape.

These weapons *are* indeed useful in DukeMatch games and should be used at least as frequently as any other weapon you make available to your players. Perhaps it is a bit harder to kill an opponent with these weapons, but then it will be all the more satisfying to do so.

Use C-9 Canisters

These exploding canisters can be used to protect coveted artifacts by surrounding them with the canisters. You will have to come up with a way to prevent a player from exploding all the canisters to free up the item, however. Perhaps place the canisters out of firing range unless the player is very close, where setting them off could be fatal.

Starting with version 1.4 of *Duke Nukem 3D*, you are able to create new CON-file code for existing actors. This will allow you, for example, to destroy artifacts when a nearby explosion is triggered. This will definitely prevent players from setting off the C-9 canisters hastily to get to a goody. Placing C-9 canisters in areas that players suddenly appear also makes for great booby traps. These areas include under water, near teleporters, and at the edge of an elevator.

Give 'em Pipe Bombs

Oh beauteous pipe bombs! These little packages of hell were obviously created with DukeMatch in mind. I'm waiting to see someone design a level where the only available weapon is the default gun and hundreds of pipe bombs. The uses for this excellent weapon are plentiful in DukeMatch levels:

- ❖ They can be dropped inside an elevator.
- ❖ They can be placed around a body of water or near a doorway.
- ❖ They can be placed in moving subway cars.

I'm sure there are dozens more uses for pipe bombs that I haven't even thought of yet, so just make sure that you put plenty of them in your DukeMatch level. The entire concept of a pipe bomb is to lay a trap for an opponent to stumble into, and like laser trip bombs, this can be the most soul-satisfying way to annihilate someone.

MORE TOYS FOR DUKEMATCH

Shooters

The shrink ray shooter makes a pretty good obstacle, especially if you allow it to be turned on and off with a nearby (or not so nearby) switch. The Build documentation states that the shooter cannot be turned off, but as you saw in chapter 7, you can create a simple shooter tied to an Activator and a Switch sprite that turns on and off just fine.

Engine Pistons or Crushers

Would it be fun to lure opponents under an engine piston and crush them paper flat? You bet it would! A series of pistons in a row evokes images of Sylvester chasing Tweety Bird around the construction site in that old cartoon. If was entertaining back when that cartoon was made (and still is), then it would be just as entertaining in your DukeMatch level. Besides, loud moving machinery can help to establish mood for your level.

Jetpacks

Jetpack battles can be extremely fun in DukeMatch. I saw one level where most of the fighting took place high above the ground, and a common way of perishing was simply having your jetpack run out of juice!

Also keep in mind the escape potential of a jetpack, especially when an opponent has just been shrunk down from a shrinker's blast. (Your opponent can't step on you if you're airborne!) Jetpacks are an easy way of adding that Z dimension to your level.

The Holoduke

This is another goody that had to be designed with DukeMatch in mind. Holodukes can lure opponents into ambush areas or provide a split second of confusion in order to make a quick exit. A great trick is to somehow convince opponents that the image before them is a holoduke, so they ignore it when it's actually a real player standing still waiting for them to get too close. Holodukes are another one of those items that have dozens of uses, so make sure a few are lying around in your DukeMatch levels. The one drawback to using a holoduke is that it can easily be distinguished as fake by a player wearing night vision goggles.

Cameras and Monitors

As the designer of your level, you'll obviously know where all the good hiding places are. Make sure to use the security cameras and monitors (ViewScreen sprites) to give some of those hiding places away. Don't make it *too* easy to find a hidden opponent, however. Maybe put the lone monitor in a hard-to-reach location, but reward the player who gets there with a bird's-eye view of most of the level. Or, use the camera and monitors as more of an *offensive* weapon, giving a well-hidden player a way to see when the opponent is approaching.

Water Sources

Water fountains, broken toilets, and fire hydrants provide water sources that are a nice cheap way to allow the players to replenish health, but their healing effects take a long time to obtain. Here is another risk and reward factor: Players get the benefit of more health if they're willing to stand around in one place long enough. To make matters worse for players, make sure to put these objects in wide open areas so the players will make good targets.

Night Vision Goggles (NVGs)

I've already mentioned that these devices are the *antidote* to the impact of the holo-duke, but their basic function is of course to allow the player to see in the dark. Once again, this allows you to set up balance in your level: Players trying to gain an advantage by hiding in a dark area will give up that advantage to a player wearing NVGs. These objects are necessary in a level that features a dark area because balance is the overall goal of the great DukeMatch level.

EXAMPLE DUKEMATCH LEVELS

The Spin Cycle level in Lunar Apocalypse (E2L10) and the Tier Drops level in Shrapnel City (E3L10) are excellent examples of many of the DukeMatch principles described in this chapter. Both are reasonably small in size, and both have at least one large common area with multiple entry points. Both are also very nonlinear in nature; there are many ways to get from point A to point B, and neither has difficult puzzles to solve.

I especially like Spin Cycle because of the large rotating sector, which causes an interesting side effect to rockets that are fired in or through the sector. Although this strange effect doesn't exactly follow the laws of physics, it adds an extra complication to a long-range rocket or devastator battle.





Using EditArt

As you might have guessed, you are not restricted to the Build-supplied graphics for creating your *Duke Nukem 3D* levels. It is possible to import graphic images that you can include in your level designs. The EditArt program, which is stored in the file EDITART.EXE in the *Duke Nukem 3D* directory, is the program that you use to import new graphics into the game. This chapter will provide a brief introduction to this program as a graphics-importing utility.

THE UTILITY OF EDITART

Unfortunately for many of us, the EditArt program is fairly difficult to use. Being a programmer myself for many years, this tool reminds me of a program that I might write strictly for my own use. When a programmer writes a utility for his or her own use, he or she doesn't pay as much attention to the minor bugs that might crop up or to creating elegant user interfaces. The tool is written to perform a certain task, which it performs.

EditArt is probably just such a program. It was written primarily as the graphics importer for *Duke Nukem 3D* and other Build engine games. There are many other features of the program that allow it to perform certain other graphics functions, similar to a paint program. However, in this chapter you will concentrate on using EditArt *only* as a graphics-importing utility for *Duke Nukem 3D*. There are so many great graphics programs on the market that you are probably better off using a graphics program with a much friendlier interface and more intuitive command structure for creating and preparing your graphics.

SETTING UP EDITART

There is one setup step that needs to be performed before importing graphics into EditArt. *Duke Nukem 3D* stores all of its graphics into files with an ART extension, and you need to extract these files from the main DUKE3D.GRP file so that you can add graphics to these files. The following command will export all of the ART files from the GRP file and put them in the main *Duke Nukem 3D* directory:

```
KEXTRACT DUKE3D.GRP *.ART
```

The extracted files will have the names TILES000.ART, TILES001.ART, and so on all the way up to file TILES014.ART. Before learning to use EditArt, you'll need some type of royalty-free graphics data that you want to import into the game. You can either create such graphics

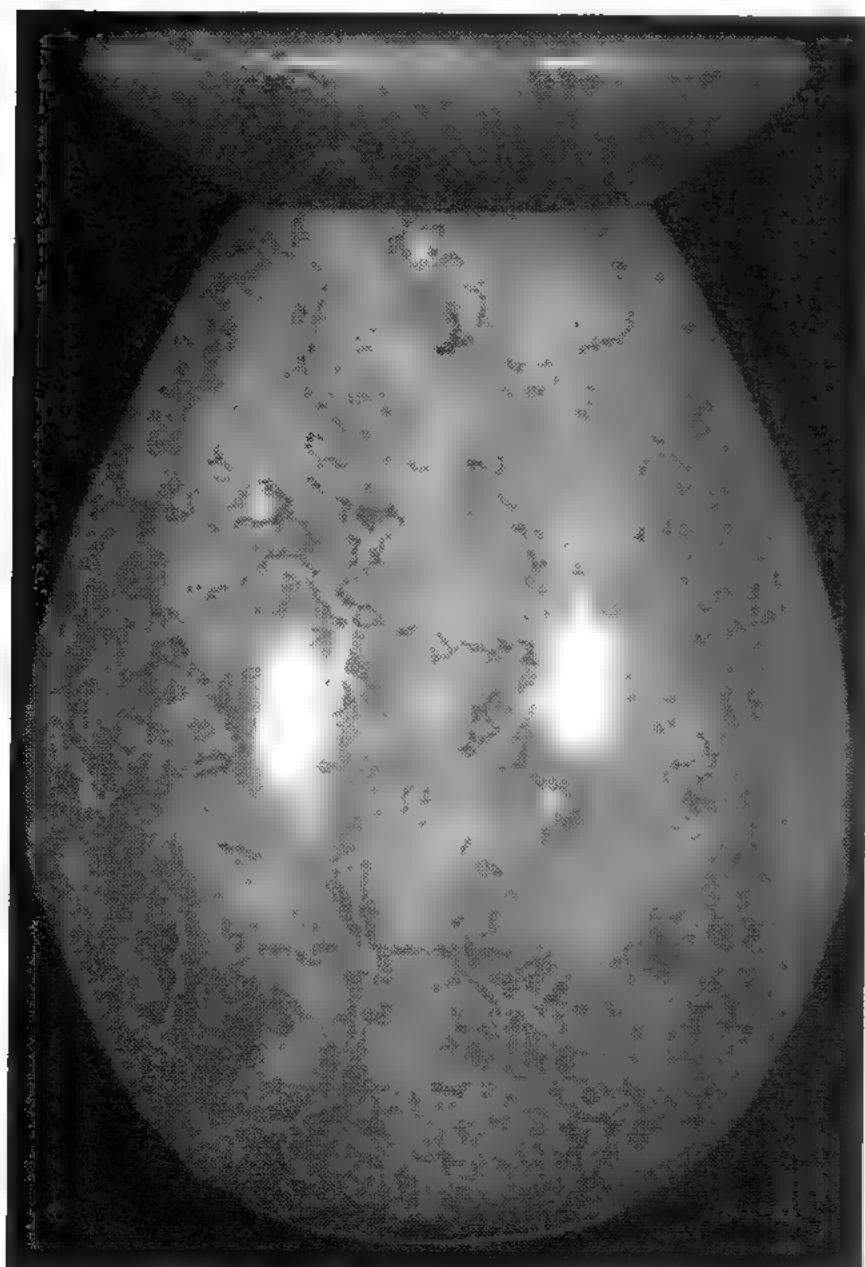
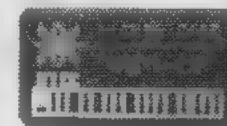


FIGURE 11.1: This vase image was created in a program called PovRay.



NOTE

One of my own graphic sources is the freeware ray-tracing program PovRay. This program allows you to write an ASCII text file known as a script that describes some type of scene, be it a single object or an entire area. The program then translates the script into a graphic of the scene described. If you are into computer graphics, I highly recommend finding a copy of this program and checking it out. PovRay can be found in the Graphics Developers Forum on CompuServe (GO GRAPHDEV).

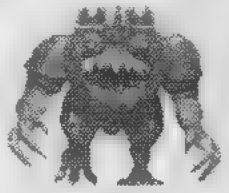
yourself, or

find one of the numerous sources of royalty-free art in commercial graphics libraries or on the Internet. In discussing the use of this utility, I'll assume that you already have some type of graphic that you want to import into the game.

I used the PovRay program to create an object that I wanted to add to the game: a simple vase. A picture of the final vase image is shown in Figure 11.1.

PREPARING THE GRAPHIC FOR IMPORTATION

The vase image you see in Figure 11.1 was much larger than the final size that I needed it

**WARNING**

I mentioned the phrase *royalty-free* in association with finding new artwork. Many shoot-'em-up game aficionados will no doubt be gung ho to import all the graphics from a certain other 3D first-person shoot-'em-up game, already mentioned several times throughout this book. My recommendation is to avoid this at all costs. The authors of this other game have been known to get quite upset at the copyright infringement of their artwork, and a lawsuit from a software company is something you probably don't need at this point in your life. This also goes for the use of artwork or characters from other copyrighted sources, such as certain movies or television shows. Blowing away cartoon characters you've seen on television sounds like great fun, but the creators of said cartoons may not see the enjoyment in it and might back up their displeasure with lawyers.

for the game. *Duke Nukem 3D*'s sprite graphics are usually smaller than 128 pixels square, and my vase was over 200 pixels square. Therefore, I needed to shrink the graphic down to size. In addition, the output associated with the particular graphics program I chose is a file format known as a Targa file (Targa files have a .TGA extension), which is a file type EditArt doesn't know about. Also, graphics imported into *Duke Nukem 3D* must all be of the same 256-color palette. Keep these points in mind if your graphics program provides other graphics file formats or can support up to 16 million colors.

The following summarizes the steps required to convert my vase image to an image that EditArt would be able to handle. I used the Paint Shop Pro program to do this conversion process.

- ❖ If necessary, crop the image so that there is as little blank border as possible without cropping any part of the image itself. This is what I did for my vase image.
- ❖ If necessary, reduce the size of the graphic down to a more suitable size. I reduced my vase to 72 × 111 pixels by choosing the width of the picture as 72 pixels, and the Paint Shop Pro graphics utility adjusted the height automatically to maintain the same aspect ratio for the image.

- ❖ If necessary, convert the graphic from its current color palette to a 256-color graphic by loading *Duke Nukem 3D*'s palette. I converted the 16 million colors my graphics utility program allows into a separate PAL file. The palette you should use for all your *Duke Nukem 3D* graphics is included in the EditArt directory on the CD-ROM in the file named DUKE3D.PAL. This file can be used within the Paint Shop Pro program to convert any picture to *Duke Nukem 3D*'s palette.
- ❖ Very likely you will need to convert the black space around the vase to the *transparent* color that's used in the game. The transparent color must be the last color in the *Duke Nukem 3D* palette. To convert the black border to color 255, I selected color 255 as the current foreground color and performed a flood fill in the areas around the vase.
- ❖ The final image needs to be saved in a format that EditArt can understand. The documentation states that EditArt can import a BMP, PCX, or GIF file. I have had good luck importing GIF files, so I saved my current image as VASE.GIF. To import a BMP or PCX file, save a 320 × 200 size picture as 256 colors.

**NOTE**

To manipulate my graphic and convert the Targa file format to one suitable for importing into *Duke Nukem 3D*, I needed a good graphics utility package that could do many different types of graphic manipulations. My tool of choice is Paint Shop Pro by JASC Software. Paint Shop Pro has always been a bargain at around \$69, as it can perform many different operations on graphics files. In addition, there is a shareware version of Paint Shop Pro, which allows you to try before you buy. Paint Shop Pro can handle all of the graphics manipulations needed to get your graphic ready for use in *Duke Nukem 3D*, and once again, I highly recommend it.

The final vase image can be seen in Figure 11.2. Obviously, some resolution of the original image has been lost because both the size and the number of colors have been reduced in the image.

IMPORTING THE GRAPHIC INTO EDITART

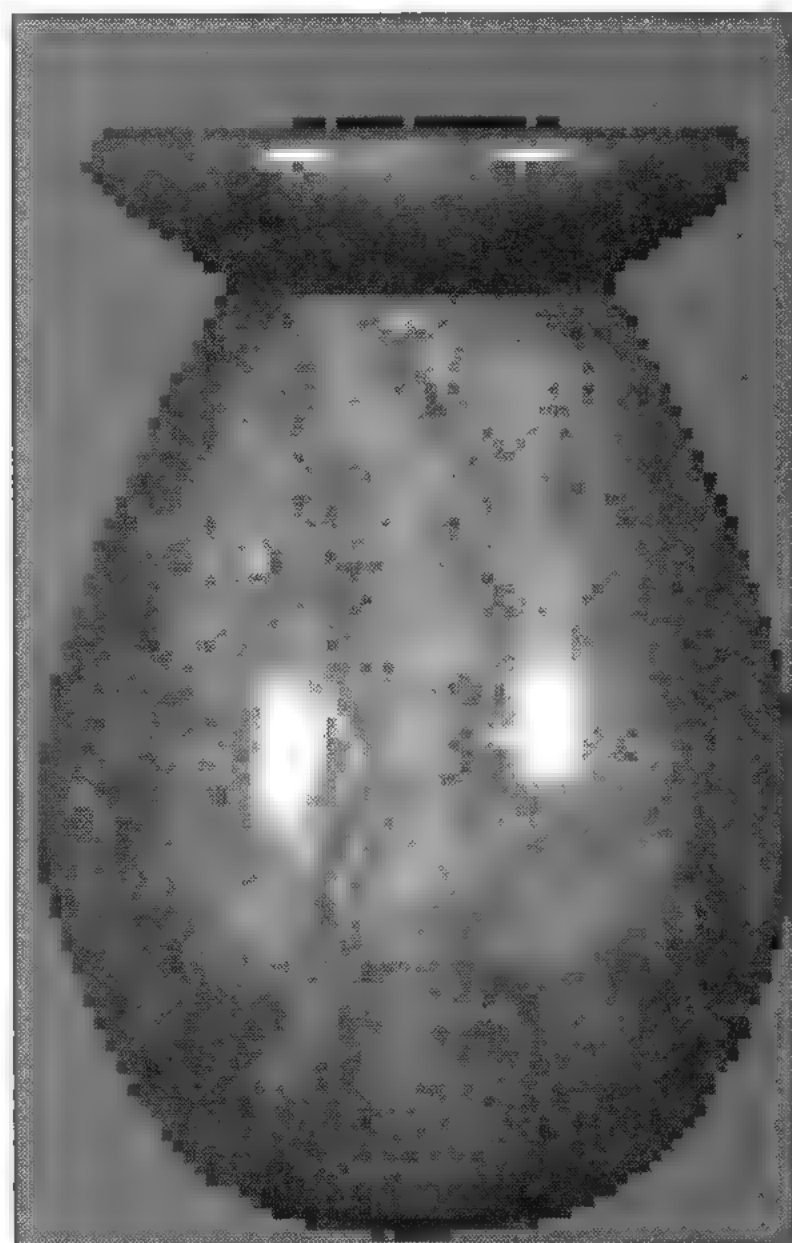


FIGURE 11.2: This vase image has been converted to GIF format with color and size reduction.

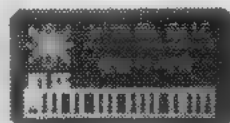
The vase graphic is now ready for importing. To start EditArt, just type EDITART from the directory containing the *Duke Nukem 3D* program files. After a brief startup screen, you should see the main editing window, as shown in Figure 11.3.

In the center of the screen, the current graphic that you are working on appears. The default graphic is the default brown tile graphic, texture #0.

To go to another graphic for altering, you can either press the G key and type the number of the graphic you wish to go to, or press the V key to bring up a graphical list of all the available textures (similar to the way you choose graphics in Build itself).

The makers of *Duke Nukem 3D* have reserved a special place in the ART files for user graphics. This place begins at graphic location 3585. You can use all the texture numbers 3585 to 4095 for your own graphics. Begin by going to location 3585 now. To do this, simply press the G key, type **3585**, and press Enter. You should now be given a blank graphic, ready and waiting for importing your own.

The main import key for Build is the U key. When you press the U key, you will be presented with a list of all the graphic files in the current directory. By using the up and down arrow keys, you should be able to find the graphic you want to import. Once the arrow is on the file you want, press Enter to select this file for importing.



NOTE

You might want to keep all your custom graphics in a new subdirectory and out of the directory containing the *Duke Nukem 3D* files. Name your directory PICS or something similar, to avoid cluttering up the *Duke Nukem 3D* directory with so many files.



FIGURE 11.3: The main EditArt screen appears when you start the program.

The next screen that comes up should show your imported picture exactly as you saved it using your graphics tool of choice. You should also see a flashing white dot or rectangle somewhere on the picture. This flashing dot is the selection rectangle for choosing the size of the graphic, which is useful if the picture you created is larger than the final picture you wanted to import. Because my picture is already its final intended size, I simply drag the mouse so that the rectangle selects the entire graphic, and press Enter. What you should now see is your picture on the main EditArt screen, imported into the spot you chose.

Loading the Imported Graphic in Build

It's now time to go check out the new graphic in Build. To do so, press Esc to exit EditArt. The program will ask if you want to save the current tile, so press the Y key to save it. This will save the graphic into file TILES014.ART or TILES015.ART, depending on the texture number into which you chose to save the graphic.

Now, start Build, make yourself a room, and place a sprite somewhere. Then, in 3D view mode, press the V key to bring up the texture selection screen, press the G key to go to a certain tile number, and type 3585, or the number for the tile your new graphic is saved under. Then, press Enter to select that tile. Your graphic should be right in your room! My vase is shown in an actual level in Figure 11.4.

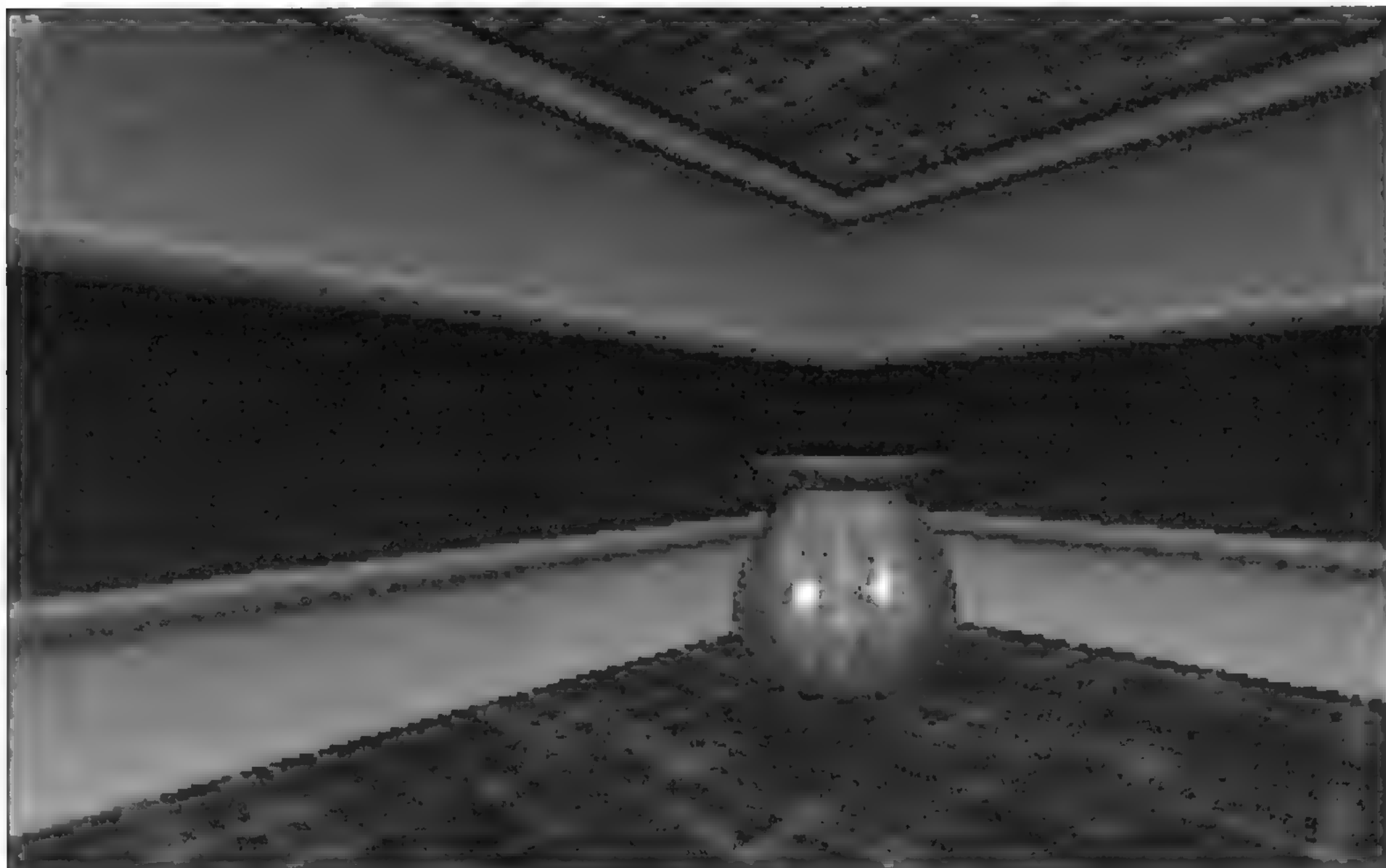


FIGURE 11.4: After importing your graphic into Build, you can apply it as a texture to a sprite in your very own level.

ANIMATING OBJECTS

EditArt also allows you to create animated textures by placing the individual frames of the animated sprite in numerical sequence in one of the ART files. For example, the frames that make up the water surface are a set of three animated frames. Because the animation data is stored right in the ART file, you can see these animations right in Build, without having to go into the game to see them. To illustrate the process of animating an object, I created an object that looked like some large mechanical piston that will pump up and down. I created three frames for this object, the first of which is shown in Figure 11.5.

The method for importing the piston object's frames was accomplished the same way as for the vase object. The pictures were created, and then reduced in size and

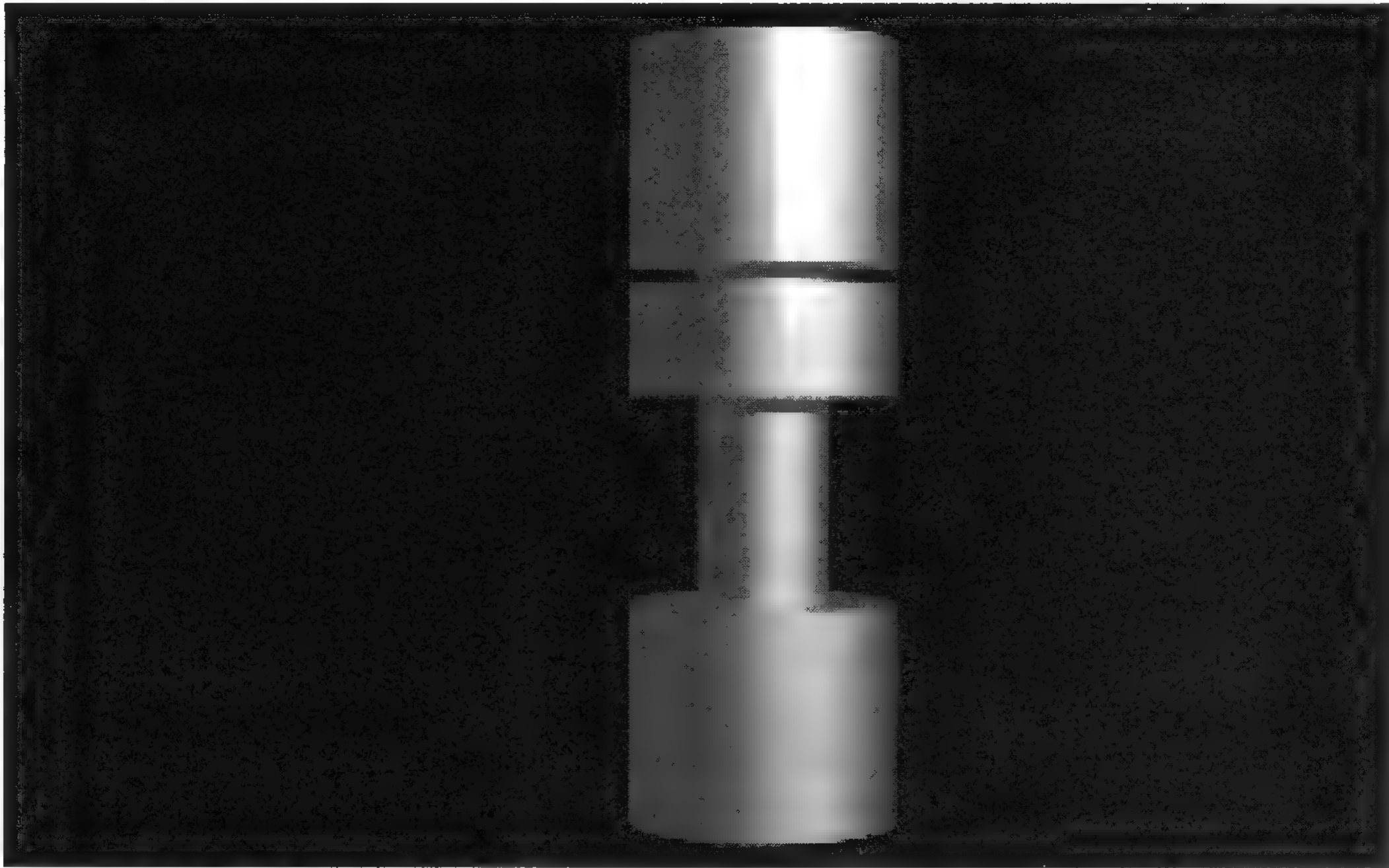


FIGURE 11.5: This is the first frame of a graphic that will be animated.

color, and finally the background was changed to color 255. A bit more care was needed this time, however, to *line up* each individual frame with the first one so that the object wouldn't shift by a pixel or two during the animation. Once each individual frame was created, I imported them into texture locations 3586, 3587, and 3588.

The next step is to define the animation. This is performed by going to the first frame in the sequence. You should see the message "Oscil: 0" in the status bar. This refers to the animation style for this object.

To change the animation, press the - (minus) key while on the first frame of the sprite. Each time you press the key, the message will toggle between four different types of animations, described as follows:

- ☢ NoAmn: No animation (0,0,0,0,0,0...) occurs.
- ☢ Oscil: Oscillating frames – The frame sequence travels forward in direction, then backward (0,1,2,3,2,1,0,1,2,3,2,1,0...).
- ☢ AnmFD: Forward – The frame sequence travels forward, starting over at 0 when the end is reached (0,1,2,3,0,1,2,3...).

- ❖ AnmBk: Backward – The frame sequence travels backward through the tiles (0,-1,-2,-3).

Note that the AnmBk animation is created by setting the animation values to the *last* numerical frame in the ART file, and the others are created by setting the *first* frame in the ART file.

Once you choose the animation style that you want, press the + (plus) key to increase the number of frames in your animation. Note that the number you choose is actually one *less* than the number of frames you have, so if you have three frames, the value you want to choose using the + (plus) key is 2. The - (minus) key is used to decrease the number of frames. When the frames are 0, the - (minus) key toggles the animation type, as explained earlier.

For the piston example, the selected animation is Oscil:2, which describes a three-frame oscillating animation that plays frames in the order 0,1,2,1,0,1,2,1,0,1,2,1, and so on. This will give the illusion of the piston pump traveling up and down.

Once the sprite's animation is defined, you can also change the speed of the animation in EditArt by pressing the A key while the first sprite frame of the animation is selected. The animation will be performed for you. Using the + (plus) and - (minus) keys will speed up or slow down the animation. Once the animation is at a good speed, press the Enter key to save this animation setting. You can place the animated graphic into a level now the same way you placed a nonanimated one. Figure 11.6 shows my piston graphic placed in a level.

If you create new graphics for your levels and want to distribute these graphics, all you have to do is include files TILES0014.ART and TILES0015.ART with your level. The other ART files won't be needed, because they contain all the original artwork.

As I mentioned before, EditArt has other functions for painting or art touch-up. If you find it easy and effective to use the program in this way, then feel free to do so. Tables 11.1 and 11.2 present all the keys and their functions in EditArt, as provided in the program's documentation. These are keys you will need to know, if you want to select from a section of a $320 \times 200 \times 256$ picture file (BMP and PCX only) and put it into the BUILD engine.

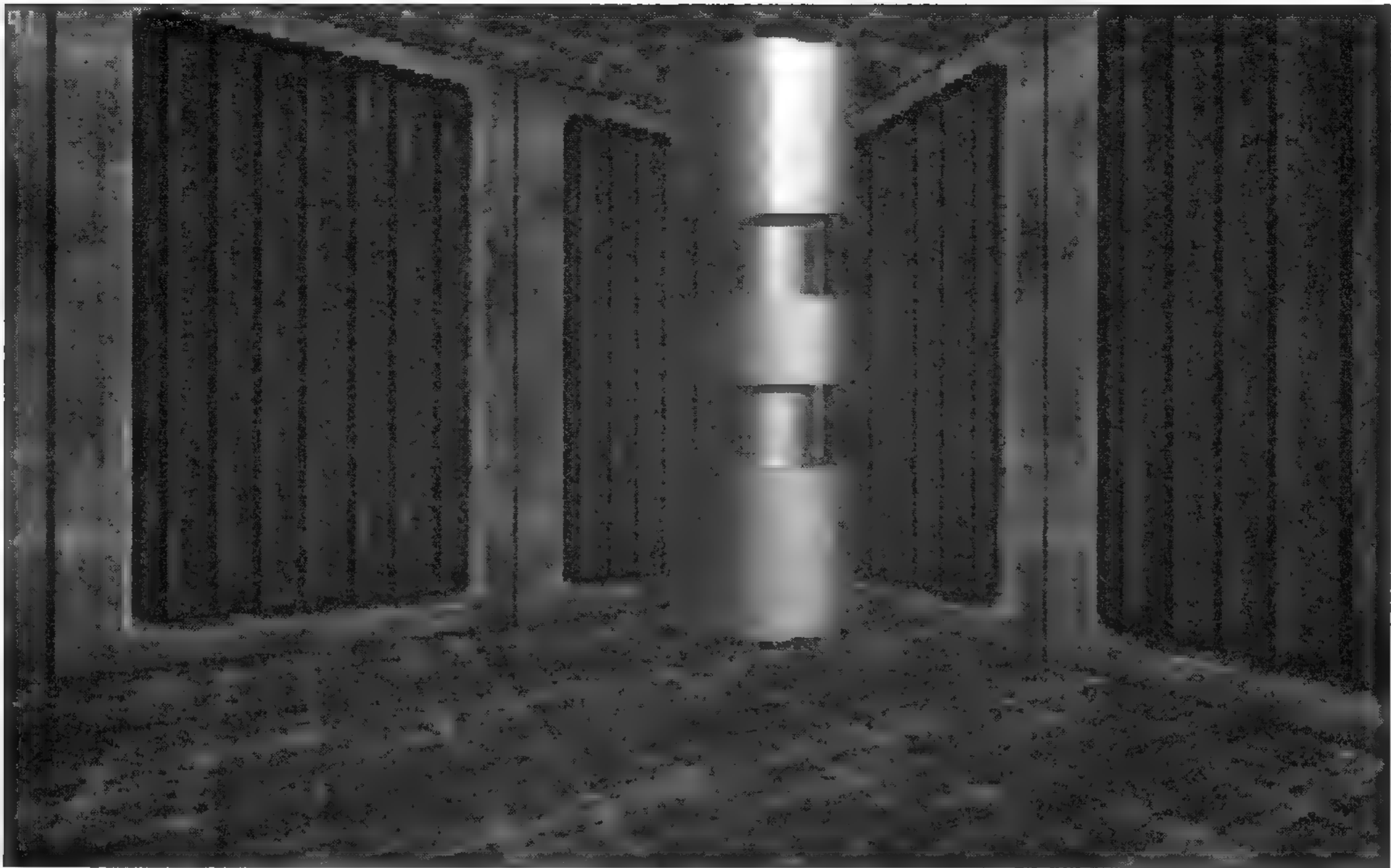


FIGURE 11.6: My animated piston graphic is shown here placed in a level.

TABLE 11.1: SELECTION AND MANIPULATION KEYS

| KEYS | DESCRIPTION |
|-----------|---|
| U | Import a section of a 320x200x256 BMP, PCX, or GIF graphic. |
| Enter | Convert the image that is inside the selection rectangle to the BUILD palette. |
| spacebar | Convert the image that is inside the selection rectangle without remapping the palette. |
| P | If in the picture selecting screen (after pressing U and loading the picture), you press P, then Build's palette can be replaced by the palette of the displayed picture. |
| PgUp/PgDn | Select tile to edit (4096 tile maximum right now). |
| G | Go to a tile by typing the tile number. |
| S | Resize a tile. The X and Y sizes can be any unsigned short integer. X ranges from 0 to 1024, and Y ranges from 0 to 240. |
| Del | Shortcut key to set both the X and Y sizes to 0. In texture selection mode, when X and Y = 0 it will delete the tile and shift all other tiles down one. |
| Ins | In texture selection mode, it will insert a blank tile space. |

(Continued on next page)

(Continued from previous page)

| KEYS | DESCRIPTION |
|-----------------|---|
| +,- (Keypad) | Change the animation setting. (Default: NoAnm = 0) To change the animation type, press the - (minus) key when the value is 0. Example: If you want an object to have four tiles of animation, you can animate it in four different sequences, as described in the text. |
| A | Set the animation speed of the tile. Press + (plus) and - (minus) keys to change the animation speed. There are 16 different animation speeds. The animation speed set here sets the speed for Build and your game. Speed is proportional to (totalclock>>animspeed). |
| ` | This key (located just above the Tab key) allows you to center a sprite. Simply use the arrow keys to get to the desired position. |
| N | Name a tile. Naming a tile simply changes the #define statement in NAMES.H. You should include NAMES.H when compiling so you can easily refer to sprites by name rather than by number. |
| O | Optimize the size of an individual piece of artwork. Use this for tiles with invisible pixels on the sides. |
| V | View and select a tile to edit. |
| spacebar | To swap two tiles, simply press spacebar on the first tile, and then spacebar on the second. |
| 1,2,3 | To swap a group of tiles, press 1 on the first tile, press 2 to remember the region between where you pressed 1 and 2. Press 3 at the place to where you want to swap all the tiles. |
| Alt+U | Regrab artwork from original pictures according to the CAPFIL.TXT file. If you press Alt+U in the main screen, everything will be regrabbed. If you press Alt+U in view mode, then you should first select the range by pressing 1 and 2 keys on the range boundaries. |
| Alt+R | Generate a tile frequency report by scanning all maps in the directory. Use in V mode only. |
| F12 | Screen capture (saves image as a BMP file, starting as file name CAPTUR00.BMP and incrementing by 1 each time F12 is pressed). |
| Esc | Quit. |

TABLE 11.2: GRAPHICS EDITING AND TOUCH-UP KEYS

| KEYS/ACTION | DESCRIPTION |
|--------------|--|
| C | Change all pixels on the tile having the same color under the graphics cursor to the selected color. |
| arrows/mouse | Move graphics cursor. |
| Shift+arrows | Select color (on bottom right corner of screen). |

| KEYS/ACTION | DESCRIPTION |
|-------------|---|
| spacebar | Plot a pixel with the selected color. |
| T | Turn drawing trail on and off. |
| Tab | Select the color under the graphics cursor. |
| Backspace | Set the color to color 255 (transparent color). |
| F | Flood-fill a region with the current color and with the current color as a boundary. |
| M, P | Use M to back up a tile into a temporary buffer in memory and P to restore it. It may be wise to press M before a flood-fill (F) because sometimes you miss encapsulating the region by 1 pixel, and the whole picture gets killed. |
| J | Randomly plots dots of current color over any pixels having the same color as the color under the tile cursor. |
| [| Random anti-alias of colors in color band under graphics cursor. |
|] | Nonrandom anti-alias of colors in color band under graphics cursor. |
| ; | Make an image 3D. Makes colors in different rows of the color bar appear to either stick out of or stick into the wall. |
| ' | Make an image 3D the other way. |
| R | Rotate the tile in a specified direction. |
| 1 | Mark the first corner of a rectangle for a copy/paste operation. |
| 2 | Mark the other corner of a rectangle for a copy/paste operation. |
| 3 | Paste the selected rectangle. (Note: You must press 1 and 2 in that order before pressing 3.) |
| 4 | Flip the copied rectangular region x-wise. |
| 5 | Flip the copied rectangular region y-wise. |
| 6 | Swap the x and y coordinates of the copied rectangular region. |
| ,.<> | Change the shade of the selected region. |
| \ | Move the cursor to the center of the tile. |
| | Get the coordinates of the cursor. |

EditArt is single tool that can make your Duke Nukem 3D levels stand out and make people really take notice, because it allows you to add objects, signs, and textures that the players of your level have never seen before. It also vastly extends the possibilities for the scenarios of your level. Want your players to travel back in time to the old west? Just set up a time travel station somewhere, add some good “old west” genre artwork using EditArt, and whip up an old west town. The possibilities are truly endless.





Editing CON Files

When you start *Duke Nukem3D*, you may notice the message “compiling GAME.CON.” The file GAME.CON, and two others with the CON extension, can be found in the directory containing the *Duke Nukem 3D* files. These three files are ordinary ASCII text files that can be viewed and edited with any text editor, such as DOS EDIT or Windows Notepad.

The purpose of these files is to define many different game play attributes of *Duke Nukem 3D*. By modifying these files, you can change the environment within the game fairly extensively. This chapter will explain the general structure of these files and ways to modify them to achieve different effects in the game.

The first step you should take before editing the CON files is to back up the original ones. Editing CON files is a fairly difficult task, and you’re bound to make a few mistakes along the way. Having a clean backup of these files is the best way to avoid any disasters, because you will always be able to start fresh regardless of how you might mess up the edited files! If you want to access the original CON files, type **kextract duke3d.grp *.con**. Consider making a directory within the directory containing your *Duke Nukem 3D* files. You can name the subdirectory *Orig* and copy *.CON to the Orig directory. Now you’re ready to go!

DEFINING NUMBERS IN DEFS.CON

The DEFS.CON file is used for redefining numbers as easier-to-read names for use in the other CON files. The basic structure of every line in this file is as follows:

```
define <name> <#>
```

Once a value is defined, the < name > can be used in its place to create more readable code. Those of you with programming experience will recognize a *define* statement as a constant declaration. For those without programming experience, just keep in mind that when you see a word in the GAME.CON file that's in all capital (uppercase) letters, this usually tells you that it is a name that was defined either earlier in the same CON file or in another CON file.

There are seven blocks of defines in DEFS.CON. Each is described in Table 12.1.

| TABLE 12.1: FIRST LINES IN DEFS.CON BLOCKS | | |
|--|-------------------------|---|
| BLOCK NUMBER | STARTS WITH | DESCRIPTION |
| 1 | define SECTOREFFECTOR 1 | Defines all the sprites used in the game. |
| 2 | define KNEE_WEAPON 0 | Defines all the available weapon types. |
| 3 | define faceplayer 1 | Defines all the basic ai types, used in "ai" commands (see "Working with GAME.CON"). |
| 4 | define NO 0 | Defines YES and NO to 1 and 0, respectively. |
| 5 | define pstanding 1 | Defines all the possible player actions. Used with the "ifp <player action#>" command (see "Working with GAME.CON"). |
| 6 | define GET_STEROIDS 0 | Defines all the available power-ups the player can pick up. |
| 7 | define KICK_HIT 0 | Defines all the available sounds used in the game. These sound numbers are then mapped to VOC files in the USER.CON file. |

CHANGING DEFINITIONS IN USER.CON

Like the DEFS.CON file, the USER.CON file is also a list of definitions. The definitions in USER.CON are differentiated from those in DEFS.CON because most of the definitions in this file are changeable by the user and will have some type of effect on the game. Most of the definition names are either self-explanatory or have comments

included. The following list provides some brief descriptions for the *Miscellaneous Game Settings* block of code in USER.CON:

- ❖ All lines beginning with *define* are settings that control monster hit points, weapon damages, maximum inventory levels, and explosion radii as well as other game settings, such as whether the security cameras are destructible, and how often Duke curses. Experiment freely with these settings, as they could have a significant impact on the game. Changing these settings could especially change a DukeMatch dramatically, for example, if the weapon damages were all lowered or made nearly equal.
- ❖ All lines beginning with *definequote* are definitions that assign the quotes that you see at the top of the screen to numbers for use in the GAME.CON file. These quotes can be easily changed, but make sure no quote is longer than 64 characters.
- ❖ All lines beginning with *definelevelname* name a level in the game and specify completion times. An example follows:

```
definelevelname 0 0 E1L1.map 01:45 00:53 HOLLYWOOD HOLOCAUST
```

This is the definition of the first level in the game. The “0 0” refers to game 1, level 1 (subtract 1 from both the level and the game to get these values). “E1L1.map” is the name of the MAP file created with the Build editor. This file is stored inside the DUKE3D.GRP file. The time “01:45” is the par time for completing the level, and “00:53” is the record time. If you change the times, make sure each uses exactly five characters, padding the minutes with “0” as shown in the example. “HOLLYWOOD HOLOCAUST” is the name of the level. The maximum number of characters you can use for the level name is 32.

- ❖ All lines beginning with *music* provide definitions. There are extensive notes in the USER.CON file for changing the MID files used by the game. If you want to change the music, just follow these directions, and copy the new MID file to the Duke directory.

- ☛ All lines beginning with *definesound* provide sound definitions. Todd Replogle at 3D Realms did a good job explaining how the sounds work right in the CON file, so see the section “ABOUT CHANGING SOUND FX” in the USER.CON file.

WORKING WITH GAME.CON

This brings you to the *meat* of the CON files: GAME.CON. This file contains hundreds of lines of program code that direct many of the objects in the game. If you have a background in programming (especially C programming), most of the contents of this file will be intuitive to you. If you do not have a programming background, then try to read the following as carefully as you can. Finally, don't be afraid to experiment, using all the existing lines of code as an example. If you've backed up the CON files (as you should have), then you shouldn't worry about making a mistake; you can always start over with new copies of the originals.

THE *ACTOR* KEYWORD

The following block of code defines the rubber garbage can that is seen throughout the cinema on the Hollywood Holocaust level (E1L1). It is a good object to start with in this discussion of the GAME.CON file because it is small and simple.

```
action RUBCANDENT 1 1 1 1 1
action RUBCAN
actor RUBBERCAN WEAK
    ifaction RUBCANDENT { ifactioncount 16 { strength 0 action RUBCAN break } }
    else ifhitweapon
    {
        ifwasweapon RADIUSEXPLOSION { state rats ifrnd 48 spawn BURNING debris SCRAP3
12 killit }
        else action RUBCANDENT
    }
enda
```

Let's skip the lines that start with *action* and start right away with the code that defines the actor. The actor name is RUBBERCAN. The actor name will correspond to

one of the sprite names in DEFS.CON. If you search for the text RUBBERCAN in DEFS.CON, you will find the following line:

```
define RUBBERCAN 1062
```

This tells you the actor RUBBERCAN starts with sprite number 1062. The next word, WEAK, is defined in USER.CON as value 5. This is the number of *hit points* an actor has. The larger the number of hit points an actor has, the tougher the actor will be to destroy.

After the actor definition line comes the code that *runs* as long as the actor exists. The first line tests a *condition*. A condition is something that is either true or false. The condition “ifaction RUBCANDENT” is true if the actor is currently performing the action named RUBCANDENT. If the actor is performing this action, then the code following the condition will be executed. If the actor is not performing this action, then the code following the “else” part of the statement will be executed.

In this example, if the actor is indeed performing action RUBCANDENT, the code inside the curly braces { } will run. Take a closer look at that code now. This block starts with “ifactioncount 16.” This is another condition. In this case, the condition is true once an internal counter associated with the actor reaches the value 16. Every time an actor changes actions, this counter resets to 0. This gives you a way of repeating the same action in sequence for a given period of time. In the example, if the internal action counter reaches 16, the actor’s strength is set to 0, and the actor changes actions to the RUBCAN action. The *break* at the end tells the code to *fall out* of the actor code for this cycle. Because there is no “else” associated with the “ifactioncount 16” phrase, any action count less than 16 will result in the actor doing nothing.

The next line is simply “else,” which is matched up with the “ifaction RUBCANDENT” condition. Another way to think of the word “else” is to mentally replace it with the word “otherwise” when you read the code. Code following an “else” is executed if the condition associated with it is *not* true (or false). A simple way of looking at this is as follows:

```
if <condition is true>
    run this code here
otherwise (else)
    run this code here instead
```

In the example code, the condition will be false if the action of the can is *not* RUBCANDENT. The code following the else immediately tests for another condition,

“ifhitweapon.” This condition is true if the actor had just been hit by a weapon that the player fired. If the actor was not hit, the condition is false. Since there is no “else” clause associated with this condition, nothing further would happen to the actor. If the actor *was* hit by a weapon, however, the next block of code tries to determine what weapon did hit it. The next condition, “ifwasweapon RADIUSEXPLOSION,” is true if the weapon was some type of explosion (by an RPG or pipe bomb). If the weapon was indeed an explosion, the next block of commands is executed. These commands are as follows:

```
state rats
```

This line *calls* (or jumps to) another piece of code somewhere earlier in the CON file named “rats.”

```
ifrnd 48 spawn BURNING
```

If a random number between 0 and 255 is less than 48 (about a 19 percent chance), then “spawn” (create) a new actor named BURNING.

```
debris SCRAP3 12
```

This line creates 12 pieces of flying debris of type SCRAP3.

```
killit
```

This command kills this actor. Finally, if the weapon that hit the actor was NOT a radius explosion, switch this actor to action RUBCANDENT. The entire actor definition is described below in pseudo-English:

```
actor named RUBBERCAN (5 hitpoints)
  if performing action RUBCANDENT {
    if performed this action 16 times { switch to action RUBCAN  }}
  otherwise
    if I've been hit {
      if the weapon was an explosion {
        destroy myself
        call the "rats" code
        create actor BURNING 19 percent of the time
        make 12 units of SCRAP3 debris fly around
      }
      otherwise (weapon was NOT an explosion)
        switch to action RUBCANDENT
    }
endactor
```

Modifying Code

Let's play with the actor definition a bit. First, change "ifactioncount 16" to "ifactioncount 64" for the RUBBERCAN actor. What should happen to the can is that the dent will stay in the can much longer than it used to when you shoot it with the pistol, shotgun, or ripper chaingun.

To test the action, save the modified GAME.CON file and start a new game in *Duke Nukem 3D*. You may also want to enable the God mode (DNKROZ) or the *give everything* (DNSTUFF) cheat code here so that the monsters don't bother you while you're testing your new rubber can code. Head for one of the rubber cans (they're located throughout the cinema), and shoot the can and see if it indeed stays dented longer.

If you want to test out some more changes on the rubber can, here are some more suggestions:

- ❖ Remove "ifrnd 48" that's associated with the "spawn BURNING" code. This will mean that the BURNING actor will always be created. Alternatively, you could just make the value after "ifrnd" larger, say 200. Keep in mind that valid values are 0 to 255.
- ❖ Change the "12" after "debris SCRAP3" to a larger number, say 48. This will make the can explode into more pieces. Make sure to enable the DNSTUFF cheat so you can fire an RPG round at the can when you are testing the explosion.
- ❖ Change the "ifwasweapon RADIUSEXPLOSION" to another weapon type. If you change it to "ifwasweapon KNEE," for example, the can will only be destroyed when you kick it!

THE ACTION KEYWORD

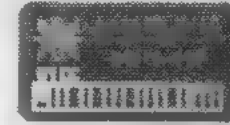
For the "action" keyword definitions, let's look at a less trivial actor and describe the actions for the octabrain. The following block of code shows the actions and the first line for this actor's definition. Starting with the actor, the name of the actor is OCTABRAIN, which is sprite 1820 (search for OCTABRAIN in DEFS.CON). It has 175 hit points (OCTASTRENGTH in USER.CON). The "fall" keyword causes the actor to fall to the floor if it is created higher than the floor.

```

action AOCTAWALK      0   3   5   1  15
action AOCTASTAND     0   1   5   1  15
action AOCTASCRATCH   0   4   5   1  15
action AOCTAHIT       30   1   1   1  10
action AOCTASHOOT     20   1   5   1  10
action AOCTADYING     30   8   1   1  17
action AOCTADEAD      38   1   1   1   1
action AOCTAFROZEN     0   1   5
...
actor OCTABRAIN OCTASTRENGTH fall
...
```

The first action is AOCTAWALK. The five numbers that follow it define an action as follows:

- ❖ The first number (1st) is the relative sprite number to start this action. This number is added to the start sprite number defined in the actor line. In this case, $1820 + 0 = 1820$ is the start sprite number for this action.
- ❖ The second number (2nd) is the number of frames that make up this action. The AOCTAWALK action is comprised of 3 frames.
- ❖ The third number (3rd) is the number of frames to skip between each successive frame of the action. In AOCTAWALK, you skip 5 frames between each frame of this



NOTE

Unfortunately, you *cannot* use saved games while testing CON files. It would be nice to do so because it would really make testing faster if you could use the game's Save Game command while testing changes to CON files. For example, you could have saved a game with Duke standing right in front of one of the rubber cans. Then you could make changes, go back into the game, and load up that saved game. Why can't you do this? The compiled CON code is part of the saved game. If you make changes to your CON files and reload a saved game, those changes will NOT be in effect. This is even more reason to become familiar with the cheat codes, especially the *warp to level* (DNSCOTTY # ##) cheat code, because it will allow you to go directly to the level in which the actor you're changing resides without having to start the game over every time. Cheat codes exist for this reason: They help developers test various parts of the game without having to play the entire game every time!

action. Therefore, the 3 frames that make up this action are 1820, 1825, and 1830.

- ❖ The fourth number (4th) is either 1 or -1. This number is multiplied to the 3rd number to allow the frame sequence to travel backward.
- ❖ The fifth number (5th) is a tempo. This is the amount of *ticks*, or game cycles, each frame will be displayed, which allows the actions to speed up or slow down.

Therefore, to get frame number $\langle n \rangle$ in an action, use the following formula, where $\langle n \rangle$ must be a number between 1 and the second number:

$$\text{Frame } \langle n \rangle = \text{StartFrame} + 1^{\text{st}} + (n-1) \times (3^{\text{rd}} \times 4^{\text{th}})$$

For another example, look at the action AOCTADYING. It is comprised of 8 frames. The first frame is 1850 (1820 + 30), and you get to the next frame by adding 1 to the current frame. Therefore, the next frames that make up this action are 1851, 1852, 1853, 1854, 1855, 1856, and 1857. This action has a tempo of 17 *ticks* per frame.

As a final actor and action example, look at the code that makes up the transporter start effect you see when someone transports in or out (actor TRANSPORTERSTAR):

```
action TRANSFOWARD 0 6 1 1 2
action TRANSBACK 5 6 1 -1 2
actor TRANSPORTERSTAR 0 TRANSFOWARD
    ifaction TRANSFOWARD
    {
        ifactioncount 6
            action TRANSBACK
    }
    else
        ifactioncount 6 killit
enda
```

Actor TRANSPORTERSTAR (1630) has 0 hit points and starts out performing action TRANSFORWARD. This action consists of the frames 1630, 1631, 1632, 1633, 1634, and 1635. (Can you explain how I got those numbers? If not, review the previous formula.) Once the actor has been in action TRANSFORWARD for 6 ticks, it switches to action TRANSBACK. This action is made up of frames 1635, 1634, 1633, 1632, 1631,

and 1630. Once this action has been in existence for 6 ticks, the actor disappears using the command “killit.” This sequence gives the effect of the transporter star growing in size, shrinking, and finally winking out of existence.

THE *MOVE* KEYWORD

The next keyword, *move*, is used to define a speed. Used on a line by itself, it can have one or two numbers after it. The first number represents movement along the floor in some direction. The second (optional) number represents up and down movement, such as bubbles moving in water. Consider the following examples:

```
move BUBMOVE          -10 -36
```

This describes a movement of 10 units in a negative direction and -36 pixels in the z direction (up).

```
move TROOPWALKVELS 72
```

This describes a movement of 72 units along the x-y plane.

THE *AI* KEYWORD

The actors now have the means to move around and to display a sequence of frames while moving, but exactly *where* will they go? Duke’s enemies seem somewhat intelligent: They can follow you around, they can hide from you, and they even avoid your shots sometimes! This *artificial intelligence* can be programmed using the *ai* keyword. The *ai* keyword allows you to tie together an actor’s action and speed with a basic ai type. These basic types, listed in Table 12.2, are defined in DEFS.CON.

TABLE 12.2: DEFINITIONS FOR BASIC AI TYPES

| BASIC AI TYPE | DESCRIPTION |
|--------------------------------|---|
| define faceplayer 1 | Makes the actor turn in the direction of the player, regardless of their distance from each other or if the actor can directly see the player. |
| define geth 2 define getv 4 | These two are used together. They mean “get horizontal” and “get vertical.” They are used for the movement of inanimate objects like bubbles and smoke. They continue the object’s movement in the same direction it is currently moving. |
| define randomangle 8 | Turns the actor in a random direction. Usually it is used for inanimate actors like jibs (flying body parts) and bubbles. |
| define faceplayerslow 16 | Turns the actor toward the player, but gradually. |

(Continued on next page)

(Continued from previous page)

| BASIC AI TYPE | DESCRIPTION |
|---------------------------|---|
| define spin 32 | Spins the actor around. |
| define faceplayersmart 64 | Causes the actor to face in the direction of the player plus the player's current rate of movement. It allows an actor to try and lead a shot, which gives it a better chance to hit a moving target. |
| define fleeenemy 128 | Causes the actor to face away from the enemy; it is used to escape or put distance between the actor and the enemy. Enemy doesn't have to be a player. |
| define seekplayer 512 | Allows the actor to perform a complicated search algorithm to find a player. The algorithm takes into account such things as players hiding around corners or at higher or lower elevations than the actor. |
| define fleeplayer 1024 | Allows the actor to flee the nearest player. |
| define looking 2048 | Causes the actor to stand still and look around. |
| define dodgebullet 4096 | Allows the actor to move to avoid an incoming projectile, as those sentry drones do! |

Using these basic ai types, you can give the actor something to do. The full syntax of the *ai* keyword is as follows:

```
ai <name> <action> <move> <basic ai type>
```

Here is the entire actor/move/ai code block for the actor known as the assault trooper (LIZARD):

```
action ALIZWALKING      0      4      5      1      15
action ALIZRUNNING      0      4      5      1      11
action ALIZTHINK        20     2      5      1      40
action ALIZSCREAM       30     1      5      1      2
action ALIZJUMP         45     3      5      1      20
action ALIZFALL          55
action ALIZSHOOTING     70     2      5      1      7
action ALIZDYING        60     6      1      1      15
action ALIZLYINGDEAD    65     1
action ALIZFROZEN       0      1      5
move LIZWALKVEL 72
move LIZRUNVEL 192
move LIZJUMPVEL 184
move LIZSTOP
```



```
ai AILIZGETENEMY ALIZWALKING LIZWALKVEL seekplayer
ai AILIZDODGE ALIZRUNNING LIZRUNVEL dodgebullet
ai AILIZCHARGEENEMY ALIZRUNNING LIZRUNVEL seekplayer
ai AILIZFLEENEMY ALIZWALKING LIZWALKVEL fleeenemy
ai AILIZSHOOTENEMY ALIZSHOOTING LIZSTOP faceplayer
ai AILIZJUMPENEMY ALIZJUMP LIZJUMPVEL jumptoplayer
ai AILIZTHINK ALIZTHINK LIZSTOP faceplayerslow
ai AILIZSHRUNK ALIZWALKING SHRUNKVELS fleeenemy
ai AILIZSPIT ALIZSCREAM LIZSTOP faceplayerslow
ai AILIZDYING ALIZDYING LIZSTOP faceplayer
```

Note that several “ai” types are defined, so the actor can do different things at different times. The next logical question would be how to program *when* the actor actually performs each ai.

THE *STATE* KEYWORD

If you have any background in computer programming, think of the *state* keyword as a subroutine. If you don’t have a programming background, then think of a state as a single block of code that’s meant to perform one task. When that task is complete, the state is said to *return* to the block of code that called it. You don’t have to use states to define your actor. The first actor example, the RUBBERCAN, didn’t use any states. Then again, this actor performed only one of two basic actions: It dented for a few seconds when shot at, or it exploded when hit with an explosion. The code to describe these two actions took only a few lines.

More complex actors (like monsters) will be performing any number of different tasks, and within each of these tasks there will be several subtasks. Consider the pig cop, for example. This monster runs after you. Occasionally, he drops to the ground and shoots at you from a kneeling or prone position, and he recoils backward when shot. He also dies. If all of these actions had to be coded in one long sequence, the code would be very difficult to read. Like all good programming languages, *Duke Nukem 3D*’s language allows you to break up a complex set of code into simpler subsets and connects these subsets by allowing you to jump from one to another by *calling* them. These subsets, then, are what’s known as a *state*. Take a look now at the full definition of the *actor* keyword, because this is where you will see the first use of a state keyword.

```
actor <actorname> <strength> <action> <state> enda
```

The `<actorname>`, `<strength>`, and `<action>` parts of the actor keyword were discussed previously. So let's look at the last part: `<state>`. If you look back to the RUBBERCAN example, notice that although there's no *state* keyword, the code that defines the actions of the rubber can happen to be located right where the `<state>` section is defined.

```
actor RUBBERCAN WEAK
    ifaction RUBCANDENT { ifactioncount 16 { strength 0 action RUBCAN break } }
    else ifhitweapon
    {
        ifwasweapon RADIUSEXPLOSION { state rats ifrnd 48 spawn BURNING debris SCRAP3
12 killit }
        else action RUBCANDENT
    }
enda
```

Therefore, the entire block of code could be called the *state* of RUBBERCAN. If you wanted to rewrite this actor using the actual keyword *state*, you could do so as follows:

```
state rubbercanstate
    ifaction RUBCANDENT { ifactioncount 16 { strength 0 action RUBCAN break } }
    else ifhitweapon
    {
        ifwasweapon RADIUSEXPLOSION { state rats ifrnd 48 spawn BURNING debris SCRAP3
12 killit }
        else action RUBCANDENT
    }
ends
actor RUBBERCAN WEAK state rubbercanstate enda
```

Note here that the state code is placed *above* the actor code because all states have to be defined *before* they are referenced, or called. Had the state and actor blocks been switched, *Duke Nukem 3D* would give an error during testing of the code in the game.

The only change to the above code that needs further explanation is the word “rubbercanstate,” which is just a *made-up* name to define this state. When calling a state, you refer to it by this name, as the above actor line does.

Now you have the means of breaking up the actor code into smaller, more manageable and readable pieces. Let’s study one such often-used piece, the famous “standard_jibs” state.

```
state standard_jibs
    guts JIBS2 1
    guts JIBS3 2
    guts JIBS4 3
    guts JIBS5 2
    guts JIBS6 3
    ifrnd 1 { guts JIBS1 1 spawn BLOODPOOL }           // spine
    ifrnd 96
    {
        ifrnd 32 globalsound JIBBED_ACTOR1
        else ifrnd 42 globalsound JIBBED_ACTOR2
        else ifrnd 52 globalsound JIBBED_ACTOR3
        else ifrnd 62 globalsound JIBBED_ACTOR5
        else ifrnd 72 globalsound JIBBED_ACTOR6
        else ifrnd 82  globalsound JIBBED_ACTOR7
        else ifrnd 92 globalsound JIBBED_ACTOR4
    }
ends
```

The *guts* command simply creates one or more copies of the sprite number that follows it. (The number of copies is the last number.) The first lines create various instances of flying body parts. Notice the 1 in 255 chance of seeing a spine (sprite JIBS1) fly away from an actor if this state is called.

After the 6 *guts* commands (including the random flying spine), there is a 96 in 256 chance (37.5 percent) that code will run to play



NOTE

This “standard_jibs” state code block is from the shareware version of *Duke Nukem 3D*. It was altered slightly in the retail version, but the shareware version is more representative of a single complete state block.

a sound. The rather large *ifrnd ... else ifrnd* block tests each case in succession, and if the random chance is met, the appropriate sound is played. Note that it is possible that *no* sound will play, even if the initial *ifrnd 96* is met and if all the *ifrnd* conditions that follow it test false.

This state is one of the most fun to play with because you can control the amount of blood and gore that the game will display. For example, simply remove the *ifrnd 1* in front of the `{guts JIBS1 1 spawn BLOODPOOL }` block, and you'll see a flying spine *every* time this state is called. Another reason I explain the functioning of this state is that it's used extensively throughout the GAME.CON file. In fact, there is a call to "*standard_jibs*" in 26 different places! This demonstrates another reason to use a state in your code: reusability. Every time the programmer wants to make body parts fly around, he or she merely adds the command "state *standard_jibs*" to the code.

CREATING NEW ACTORS

As of version 1.4 of *Duke Nukem 3D*, you can create new actors that didn't exist before in the CON files. In versions 1.3 and prior, this was not possible. This means that you can create new actions for previously inanimate objects, or you can create all new actors with your own graphics.

ADDING CODE TO AN EXISTING SPRITE

Many of the original *Duke Nukem 3D* actors have actions that are hard-coded. This means that if you were to try and write CON-file code for an existing actor, the game engine would ignore this code, and the hard-coded actions would be performed instead. There is a simple trick to get around this problem, however. There's nothing stopping you from making an all *new* object, using the graphics from the original object whose actions you want to change. The player won't be able to tell the difference between the two actors because you will use the same graphics for both the original sprite and the new one.

To demonstrate creating new actor code for an existing object, you'll use the sprite BOTTLE3, which is texture #956. The new action you'll create for this actor will be a simple (but potent) one: Make the bottle explode in a fireball when hit with a weapon.

Because the BOTTLE3 sprite's action is hard-coded to simply shatter when hit by a weapon, the first thing that you need to do is copy the BOTTLE3 sprite graphic to

another location. To do this, go into EditArt, go to sprite 956, and press F12. This exports the current graphic to a PCX file named CAPT0000.PCX in the current directory. The next time you export this way, the new file name will be CAPT0001.PCX.

In EditArt, go to an empty graphics location in the TILES014.ART file (for example, location 3589). To import the CAPT0000.PCX file, press the U key, choose this file name from the list that appears, and press Enter. Then, use the mouse to select this entire graphic so the flashing rectangle completely surrounds it, and press Enter again. This will import the graphic into the current location. This gives you an exact duplicate of BOTTLE3, but because it's a new object that the game doesn't know about, you can write new actor code for it in the GAME.CON file.

Go into the GAME.CON file, and add the following block of code to the end of the file:

```
define MYBOTTLE3 3589
useractor notenemy MYBOTTLE3 WEAK
  ifhitweapon
  {
    ifrnd 128
    {
      hitradius 512 WEAKEST WEAK MEDIUMSTRENGTH TOUGH
      spawn EXPLOSION2
      sound PIPEBOMB_EXPLODE
      killit
    }
    else
    {
      lotsofglass 8
      sound GLASS_BREAKING
      killit
    }
  }
enda
```

The first line defines a name for this new actor, MYBOTTLE3. The number 3589 is of course the same number under which you saved the graphic in EditArt. The next block looks very much like an actor block, with the exception that the keyword “useractor”

is used in place of the keyword “actor,” and the extra keyword “notenemy” is used to define the type of actor. This second parameter can be one of the following:

- ❖ *notenemy* to define a stationary actor
- ❖ *enemy* to define a new monster that will follow the player around
- ❖ *enemystayput* to define a new monster that will remain in the sector it starts in

This actor code itself is fairly straightforward. The new actor is defined using the *useractor...enda* block. The sprite MYBOTTLE3 tells the compiler which sprite this actor code describes. WEAK defines the strength of the actor; you want your bottle actor to explode easily.

The entire actor code consists of a single *ifhitweapon* block. This means that if the actor is *not* hit with a weapon, no action is performed. If the actor is indeed hit with a weapon, however, then a random test is performed. Fifty percent of the time the bottle breaks normally, with a call to “lotsofglass” so that the sound GLASS_BREAKING plays. The other 50 percent of the time, however, the code to create a large explosion is executed. This includes the “hitradius” command, the actor EXPLOSION2 being spawned, and the sound PIPEBOMB_EXPLODE being played.

Your final result is a sprite that looks exactly like the BOTTLE3 sprite, except players will find a nasty surprise if they go around firing lead indiscriminately. They just might find themselves caught in a fireball!

ADDING A NEW ENEMY

The new *useractor* statement will also allow you to create entirely new monsters for a player to battle. If you had the artistic ability or the resources to do so, you would start the process by creating all of the individual frames needed for your new creature. Todd Replogle, lead programmer of *Duke Nukem 3D*, helped me out a great deal to understand the new actor code, so I could in turn explain it to you. He even supplied me with a new sample monster to try out. Since version 1.4 of the game had not been released by the time this book was published, I don’t know if the new version will include this new monster example. Therefore, I am going to demonstrate the example here.

The actual graphics Todd used to create the monster were simplistic, to say the least; the creature consisted of a three-frame glowing sphere with the numbers 1, 2,

and 3 in each frame (he's a programmer by profession, not an artist, so cut him some slack). Simple as they were, the graphics were plenty to demonstrate how you could create an all-new monster. The first step requires importing the three graphics frames into the file TILES014.ART. Once this is accomplished, the GAME.CON code was created to bring the monster to life. This code is as follows:

```
// * Actor 'MYENEMY' is a GRAY PULSING BALL that bounces around at random
//   angles, while shooting at the closest player.
// * 'MYENEMY' causes damage when a player is close.
// * When 'MYENEMY' is hit, scrap is spawned.
define MYENEMY 3590                                // Position in .art file
define MYENEMY_NORMAL_STRENGTH 100
define MYENEMY_TOUGHER_STRENGTH 200
define MYENEMY_DAMAGE_TO_PLAYER -20
define MYENEMY_ROAM 309
define MYENEMY_HURT 310
define MYENEMY_DEAD 311
define MYENEMY_SHOOT 312
definesound MYENEMY_ROAM my_eroam.voc 0 0 0 0 0
definesound MYENEMY_HURT my_ehurt.voc 0 0 0 0 0
definesound MYENEMY_DEAD my_edead.voc 0 0 0 0 0
definesound MYENEMY_SHOOT my_eshot.voc 0 0 0 0 0
action MYENEMY_ANIMATIONS 0 3 1 1 4
move MYENEMY_SPEED 64
ai AIMYENEMY_BOUNCE MYENEMY_ANIMATIONS MYENEMY_SPEED geth randomangle
useractor enemy MYENEMY
    ifai NO // 'NO' is defined as '0'
    {
        // Prepare 'MYENEMY' for battle...
        // Give 'MYENEMY' strength. If it
        // has a palette lookup other than
        // 0, make it twice as strong!
        ifspritepal 0 strength MYENEMY_NORMAL_STRENGTH
        else strength MYENEMY_TOUGHER_STRENGTH
        sizeat 48 48
        cstat 257 // Force actor to block
        ai AIMYENEMY_BOUNCE // Make it go!
    }
```



```
ifrnd 8
{
    shoot FREEZEBLAST
    sound MYENEMY_SHOOT
}
ifcount 48 ifrnd 16
    ai AIMYENEMY_BOUNCE
ifpdist1 1024 ifrnd 16
{
    sound DUKE_GRUNT
    palfrom 24 24
    addphealth MYENEMY_DAMAGE_TO_PLAYER
}
ifhitweapon // Was it hit by a weapon?
{
    debris SCRAP1 2 // OUCH! Make some pieces fall off
    ifdead // Is it dead yet?
    { // YES?
        addkills 1 // Add 1 kill to player score
        spawn EXPLOSION2 // Make an explosion
        sound MYENEMY_DEAD // The sound of death
        hitradius 2048 WEAKEST WEAK MEDIUMSTRENGTH TOUGH // Inflict on those near
        killit // Delete sprite (CON CODE ABORTS HERE)
    }
    sound MYENEMY_HURT // No?, Make a hurt sound instead
}
enda
```

The code starts with a few define statements that locate the monster starting at sprite 3590. Then, a few hit point values are defined to create two different strengths for the monster, MYENEMY_NORMAL_STRENGTH and MYENEMY_TOUGHER_STRENGTH. Finally, a damage amount defines how much damage the creature causes.

The next block of code defines some sounds for the creature to make. These sounds weren't supplied, so I'm not sure what they are, but the code shows you how easy it is to add new sounds for your monster to make. All you have to do is create the define-sound lines as above, and include the necessary VOC files with your distribution.

Next, the monster's *action* is defined. A simple animation passes through all three frames of the creature in sequence, and a "move" command is defined to describe the monster's speed of movement. Then an "ai" statement is created to tie the monster's action with a movement, which is in random directions using the "randomangle" movement modifier.

Finally, the actor code itself begins. User actors are created in a slightly different way than other actors. When a user actor is first initialized, it has no "ai" block associated with it. Therefore, an extra block of code is required to set up the actor, as follows:

```
ifai NO                                // 'NO' is defined as '0'
{
    // Prepare 'MYENEMY' for battle...
    // Give 'MYENEMY' strength. If it
    // has a palette lookup other than
    // 0, make it twice as strong!
    ifspritepal 0 strength MYENEMY_NORMAL_STRENGTH
    else strength MYENEMY_TOUGHER_STRENGTH
    sizeat 48 48
    cstat 257                          // Force actor to block
    ai AIMYENEMY_BOUNCE                // Make it go!
}
```

The "ifai NO" test returns true if the actor is currently not performing any ai command. This will be true at the very start of the actor's existence. Within this block, the actor is given strength (weaker if the sprite has a palette of 0; stronger if it has any other palette), sized to the correct size using the new "sizeat" command, and made solid by turning on its Blocking and HitScan bits. Finally, the ai defined earlier (AIMYENEMY_BOUNCE) is enabled for the actor, which will start him on his merry way.

The rest of the actor code is fairly straightforward. The actor will move along at a random angle, changing direction occasionally. At random intervals, the monster will fire a FREEZEBLAST at the nearest player. Additional code makes the monster hurt a player too close to it. Finally, a code block is created that will cause the monster to spawn debris when shot and explode when killed.

I can't wait for some enterprising level designer to create some all new monsters with which a player can do battle. The new *useractor* is extremely powerful, and it will allow you to change the game in myriad new ways.



TS



WAD2DUKE,
a New DOOM WAD-to-Duke
MAP Converter

The retail version of *Duke Nukem 3D* comes with a program that converts levels created for *DOOM* and *DOOM II* into levels that you can play in *Duke Nukem 3D*. This program is called WAD2MAP, and it can be found in the \GOODIES\WAD2MAP directory on the CD-ROM included with this book. The idea of such a level conversion program is that all of the textures and objects in *DOOM* level can be changed into an equivalent texture or object in *Duke Nukem 3D* game, and you can then play old *DOOM* level in *Duke Nukem 3D*.

Furthermore, because *Duke Nukem 3D* offers features that the *DOOM* engine does not offer—such as floors above floors, realistic underwater areas, and moving sectors—you could take the converted *DOOM* level and add these types of effects to further enhance the level.

Unfortunately, the WAD2MAP program does not work very well. The primary reason is that the program seems to work only on the original retail *DOOM* WAD files, which means that none of the thousands of user-created PWADS can be converted easily.

INTRODUCING WAD2DUKE

Included on the CD-ROM that came with this book is a new WAD-to-MAP converter utility, named WAD2DUKE. This program was written in Borland Delphi 1.0, and it requires Microsoft Windows to run. Using this utility, you will be able to convert *all* WAD files from *DOOM*, *DOOM II*, *Heretic*, or *Hexen* into *Duke Nukem 3D* levels. This increases the number of levels available for you to play at least one thousandfold.

When you first run WAD2DUKE, you will be prompted for the location of a WAD file. You can choose any WAD file to start. Once you choose a WAD file, you will be shown the main WAD2DUKE screen, which is shown in Figure 13.1. The list box to the left of the screen lists the contents of the current directory. To change directories, click the button labeled “...” immediately above this list box.

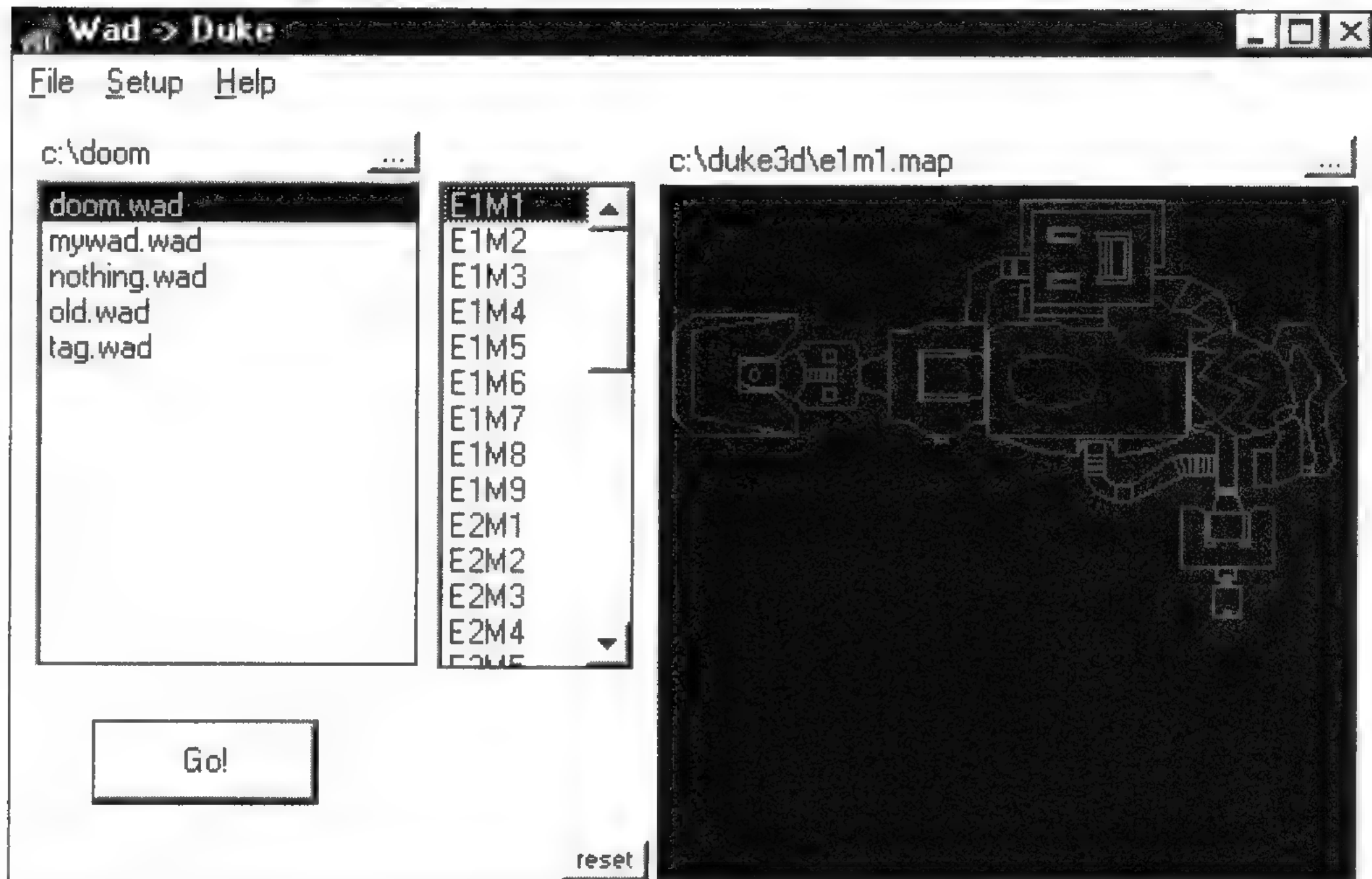


FIGURE 13.1: This main WAD2DUKE screen appears when you start the program.

The second list box shows all of the current levels located inside the WAD file selected in the first list box. In Figure 13.1, for example, you can see that the main file DOOM.WAD has been chosen, and all of the levels E1M1, E1M2, and so on are displayed in the second list box.

The map at the right of the screen shows the current level selected in the second list box. By selecting different WAD files and different levels within each WAD file, the map will change to show you the current level selected.

Above the map is the output directory. You should change this so that it is set to be your main *Duke Nukem 3D* directory. All of the converted levels will be saved to this directory.

Once you have selected the level you want to convert using the list boxes, click the big Go! button and the conversion will take place. The level will be saved as the file name shown above the map.

HOW WAD2DUKE TRANSLATES LEVEL DATA

In order for WAD2DUKE to successfully translate the level data in the *DOOM* WAD file, it needs a series of *translation tables*. These tables provide a mapping mechanism so that WAD2DUKE knows how to convert each wall, ceiling, and floor texture and object.

There are three translation tables stored in two Windows-style INI files. The names of these files are TEXTURE.INI and THNGMAP.INI. The first file, TEXTURE.INI, handles the mapping of wall, floor, and ceiling textures. The following is a sample from the beginning of the TEXTURE.INI file:

```
[Wall]
Other=0
AASTINKY=790
ASHWALL=790
ASHWALL2=790
ASHWALL3=790
```

Note the first entry, “Other,” is the texture that will be used if any *DOOM* texture cannot be found in the INI file. The rest of the entries list all the wall textures from *DOOM II* on the left-hand side of the equal sign and a corresponding texture number from *Duke Nukem 3D* on the right-hand side. This early version of WAD2DUKE does not have a good texture translation table for wall textures, so most of the wall textures are all mapped to the default texture.

The second section in TEXTURE.INI handles the mapping of ceiling and floor textures. Here are some sample entries from that translation table:

```
[FLAT]
FLOOR1_1=898
FLOOR4_8=417
FLOOR5_1=876
FLOOR5_2=1218
```

This translation table is identical in structure to the first. To the left of the equal sign lies all the available ceiling and floor textures from *DOOM II*, and on the right are the corresponding textures to be used in *Duke Nukem 3D*.

The third translation table is found in the file THNGMAP.INI. This table tells WAD2DUKE how to convert each object (called a *Thing* in the *DOOM* vernacular) found in a *DOOM* level into a corresponding *Duke* sprite. Here is a section from this INI file:

```
[9]
Desc=Monst_Sarge
Category=monst
Pic=1680
Pal=21
xRepeat=48
yRepeat=40
```

Each section in this INI file starts with a number in square brackets. Each of these sections corresponds to one type of *DOOM* Thing. *DOOM* Things are identified by a number, and the number inside the square brackets tells you which type of *DOOM* Thing this mapping is describing. In the example section above, Thing 9 is the Thing mapping being described.

The next two lines “Desc = Monst_Sarge” and “Category = monst” exist simply as comments for your own purpose, giving you a description of this particular *DOOM* Thing. In this example, the Thing being described is the shotgun-wielding sergeant.

The remaining lines describe the type of *Duke Nukem 3D* sprite that will be created in the place of this *DOOM* Thing. The “Pic” entry represents the texture that will be assigned to this actor. Because the texture of a sprite determines what type of sprite it is, this field determines what object will be placed in your MAP every time a *DOOM* Thing of the type described in the square brackets is found. In the example, texture 1680 is mapped to Thing 9. Texture 1680 happens to be the assault trooper sprite.

The Pal entry gives the sprite a palette. This field can be left blank, which will leave the mapped object as palette 0. In the example, all *DOOM* Thing 9s are given a palette of 21. It so happens that an assault trooper with a palette of 21 becomes an assault commander, with the ability to teleport and fly with a jetpack.

The next two fields, “xrepeat” and “yrepeat,” denote the size of the sprite in pixels. Most of the sprites can be left to their default xrepeat and yrepeat sizes.

One other value you can assign to a mapped sprite is the cStat value, which contains flags for making the sprite transparent, blockable, and other attributes. The following table shows you the effect of setting the cStat value.

TABLE 13.1: ACTOR CSTAT VALUES

| TK | TK | TK |
|-----------|----------------|---|
| bit 0 | (value 1): | turn on the blockable bit |
| bit 1 | (value 2): | make actor translucence |
| bit 2 | (value 4): | x-flip actor |
| bit 3 | (value 8): | y-flip actor |
| bits 5-4: | | 00 = sprite will always face actor (except mulit-frame monster sprites) |
| | (value 16) | 01 = sprite will face a single direction, usually along a wall. Often used for posters, wall hangings, etc. |
| | (value 32) | 10 = sprite will face a single direction, parallel to the ceiling or floor. |
| bit 6 | (value 64): | One-sided sprite |
| bit 7 | (value 128): | Center sprite at actual center. Default is to center along the bottom |
| bit 8 | (value 256): | Turns on HitScan bit. |
| bit 9 | (value 512): | Translucence reversing, 0 = normal |
| bit 10 | (value 1024): | Unused |
| bit 11 | (value 2048): | Unused |
| bit 12 | (value 4096): | Unused |
| bit 13 | (value 8192): | Unused |
| bit 14 | (value 16384): | Unused |
| bit 15 | (value 32768): | Make sprite invisible |

To achieve a desired effect, add up the desired bits' values in the "value" column. For example, to make a sprite invisible, use cStat 32768. To make it visible again, use bits 0 and 8, adding values 1 and 256 to get a cStat value of 257. This turns on the Blocking and HitScan bits.

All three of these translation tables are stored in standard Windows INI file format, which means that they can be edited by any text editor like Windows Notepad or DOS EDIT. Feel free to change the mappings of any or all of the Things and textures.



FIGURE 13.2: This is the opening view when playing Lost Episode E1M3 in *DOOM*.

A WAD2DUKE LEVEL CONVERSION DEMONSTRATION

Let's go through the conversion process for one *DOOM* level and see an example of how you might transform one of your own *DOOM* WADs into a *Duke Nukem 3D* level. The level that I've chosen to convert is from *The Lost Episodes of DOOM*, a book and series of *DOOM* levels written by Chris Klie (Sybex, 1995). The level shown here is Lost Episode E1M3. Figure 13.2 shows the opening view from the *DOOM* level. Note the yellow keycard up on a platform.

The first step to converting any *DOOM* level is to of course run WAD2DUKE on the level. Figure 13.3 shows the WAD2DUKE screen right before the conversion process takes place. Note that the file chosen is *LOSTE1M3.WAD* and the level chosen within that WAD file is E1M3 (the only level available in this file). Once this level has been

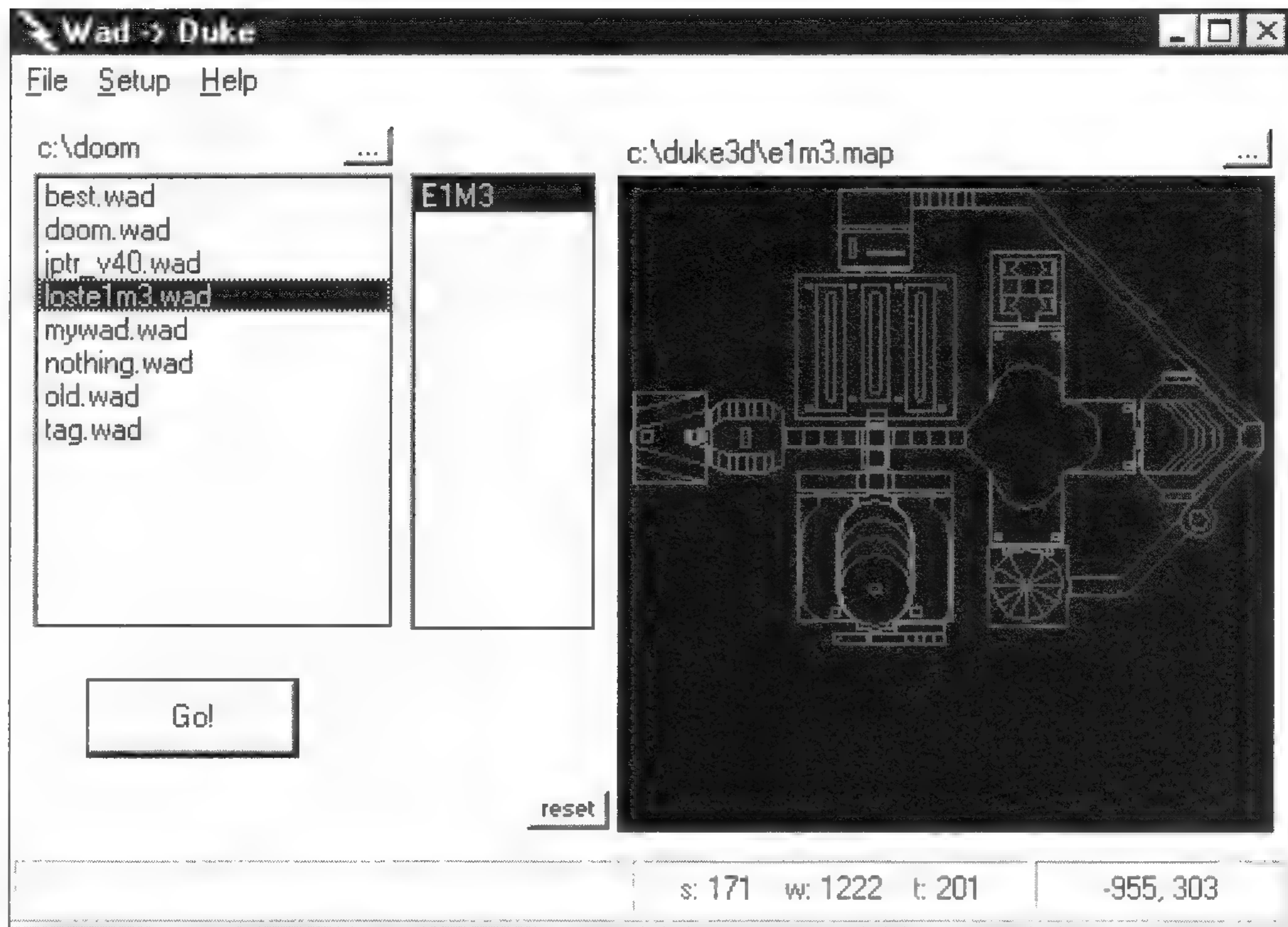


FIGURE 13.3: This WAD2DUKE screen is set up to convert the file **LOSTE1M3.WAD** to **LOSTE1M3.MAP**.

run through WAD2DUKE, the next step is to go take a look at the conversion in Build. To do this, locate the level that WAD2DUKE created and load it into Build.

While WAD2DUKE does a good job in translating the basic structure of the level, and converting all the *DOOM* Things into equivalent *Duke Nukem 3D* sprites, there are a few things you will need to complete before the *Duke Nukem 3D* version of the level is ready for action. Because you will need to make additional changes to the level, a very solid knowledge of all the features of the level you are trying to convert is an absolute must. You will especially need to know the locations of all the LineDef triggers in your level because many of these will need to be recreated in the *Duke Nukem 3D* version.

DOOM and *Duke Nukem 3D* handle moving sectors in very different ways. First and foremost, almost every *DOOM* sector action is triggered by either stepping on or pressing a LineDef. In *Duke Nukem 3D*, however, LineDefs (called Walls in *Duke Nukem 3D*) are *never* used for defining moving sectors. Instead, sector LoTags and SectorEffectors make up the majority of moving sectors.

MODIFYING TRANSLATION RESULTS

Now let's look at a few areas of the level and see what types of improvements and considerations need to be made before the old *DOOM* level is a state-of-the-art *Duke Nukem 3D* masterpiece.

After the initial conversion, it's best to simply start at the first room (where the player starts) and work your way through the level area by area. Because there are some differences in playing the two games, areas that were created to be very difficult in *DOOM* may be rendered very easy in *Duke Nukem 3D*.

One good example of this can be found right at the start of the level being discussed here. The yellow keycard is resting on a platform, which the player needs to lower in *DOOM* to obtain the keycard. However, in *Duke Nukem 3D* the keycard can be easily obtained simply by jumping up onto the platform! This consideration never needed to be made by the original designer of the *DOOM* level because the *DOOM* player is unable to jump.

Your first inclination might be to simply raise the platform a bit higher to prevent the player from jumping onto it, but even this might fail to be a valid deterrent to the player getting the key. What if the player has a jetpack? For the *DOOM* designer, placing objects up high, out of a player's reach, is a very common method of showing the player a goal that will need to be accomplished, but not right away. This same level design method is not nearly as effective in *Duke Nukem 3D*, however, because of the player's ability to jump and fly.

Therefore, areas that follow this *DOOM* design methodology will have to be redesigned to be effective in *Duke Nukem 3D*. One possible way to redesign the area in this example is to surround the platform with a force field. This will prevent the player from immediately obtaining the keycard, and because force fields can be shut off with switches, you can place the switch sprite in an identical location to where it was in the *DOOM* map.

If you look at the same area in the file LST2E1M3.MAP, you will see how a few changed details in this first area do not change the overall flow of the level. The keycard is now protected by a force field, as shown in Figure 13.4. The switch that turns off the force field is in the same location that the switch was in the original *DOOM* level, so the player will still have to overcome the same basic obstacles before getting the yellow keycard. Notice I also felt like the long windows on the west and east sides of this room were just begging for some glass, so I added this as well.



FIGURE 13.4: The yellow keycard is now protected until the player is able to deactivate the force field.

BEYOND THE YELLOW DOOR

The area immediately after the yellow door features a long hallway running west to east. The yellow door opens into the center of this hallway. To move either east or west, a small lift was created on either side of the entryway.

Here is another good example where a necessary feature built into the *DOOM* level is unnecessary in the *Duke Nukem 3D* level. Both of the lifts in the *DOOM* level raised the player only a very short height. The *Duke Nukem 3D* player can simply jump up to the higher levels of this hallway. For this reason, you don't even have to bother recreating the two lifts. Figures 13.5 and 13.6 show this area in both *DOOM* and *Duke Nukem 3D*.

THE SMALL COMPUTER AREAS

The next two rooms that you will encounter in this level are small areas filled with flashing lights and computer screens. To duplicate the walls of this area, I simply chose some of the computer panels found in *Duke Nukem 3D*. For the flashing lights, I set

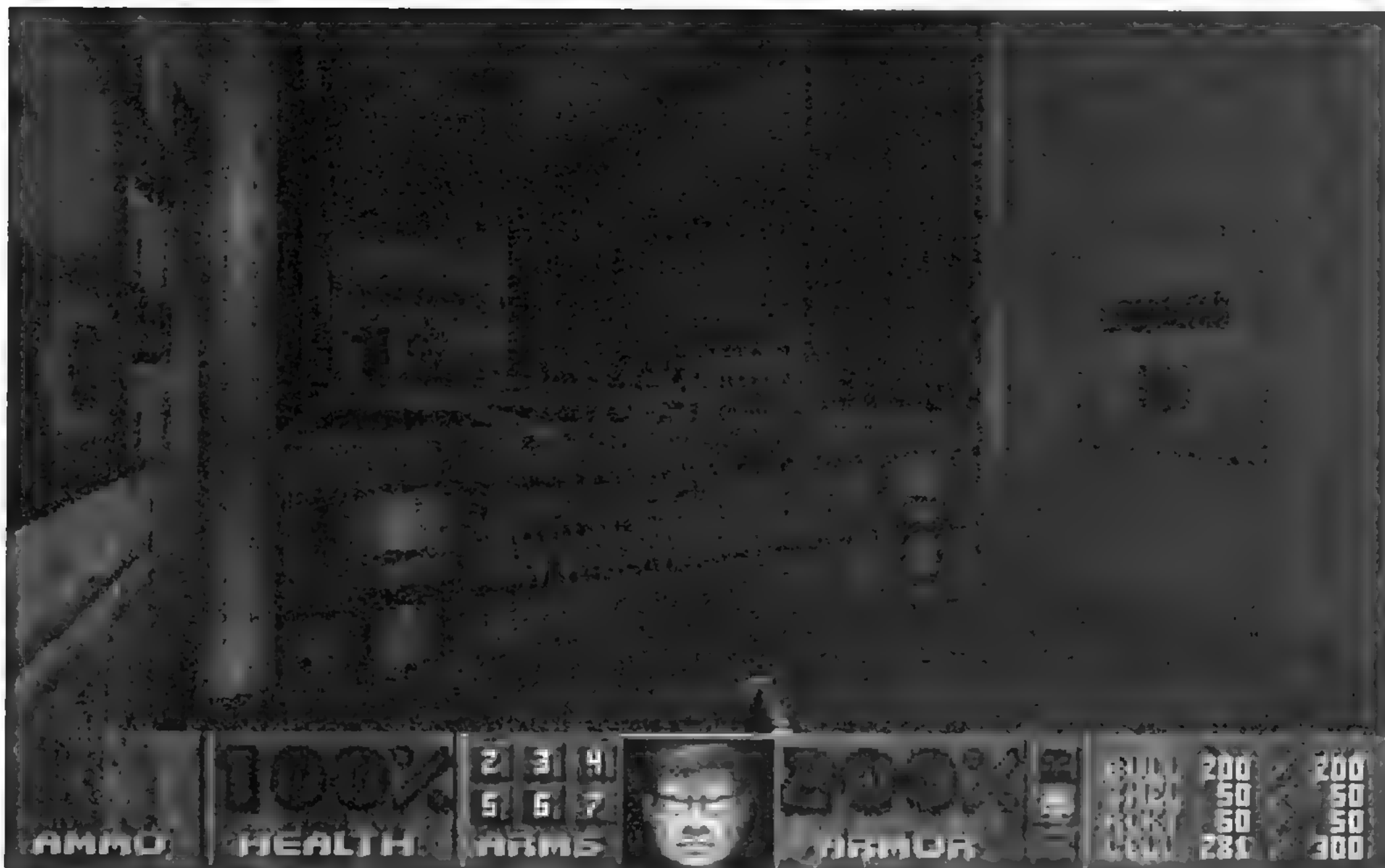


FIGURE 13.5: In the long hallway area in *DOOM* the lift is required to raise the player up to

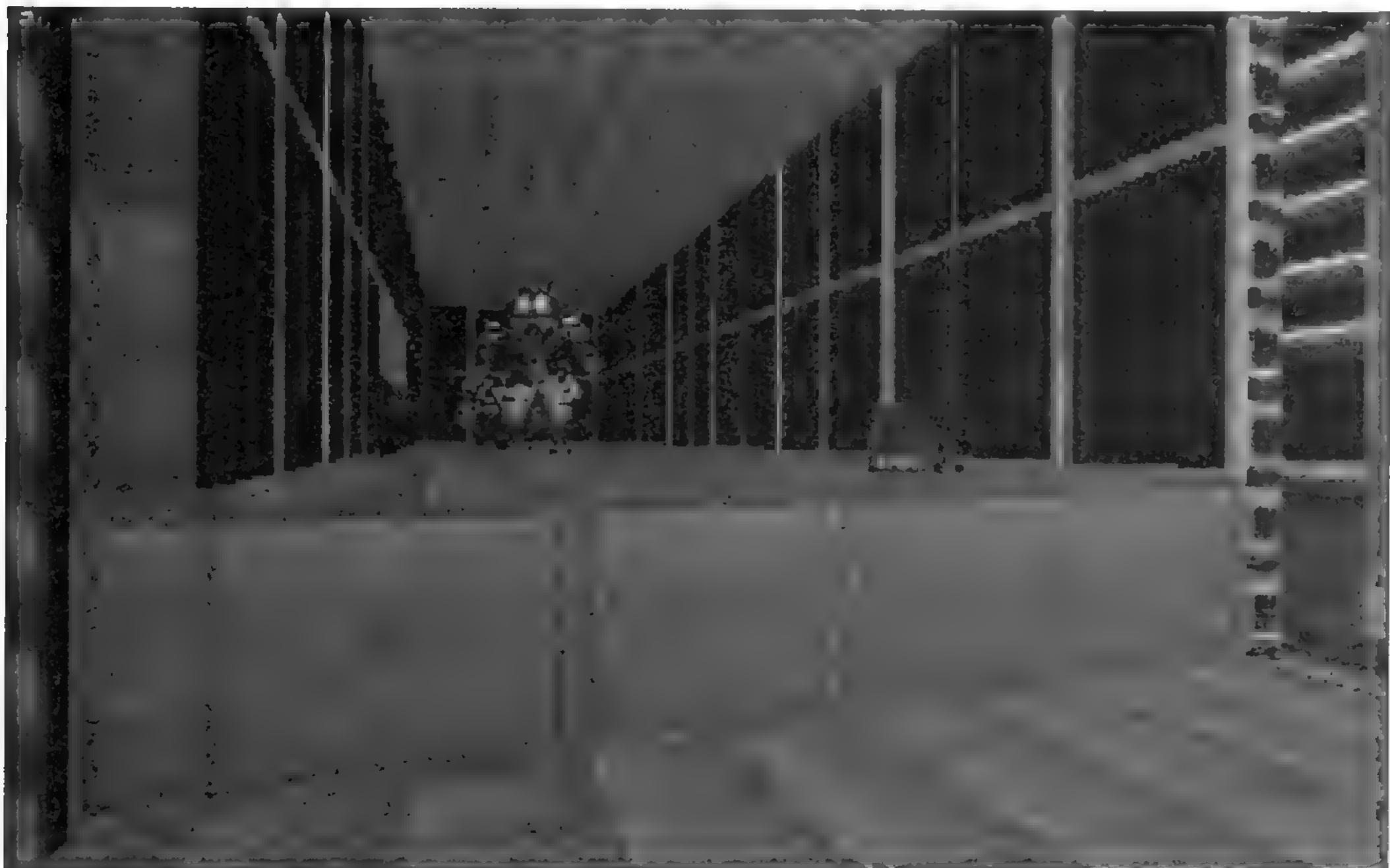


FIGURE 13.6: The lift is unnecessary in the same long hallway area in *Duke Nukem 3D* because the player can simply jump up to the hallway's floor level.

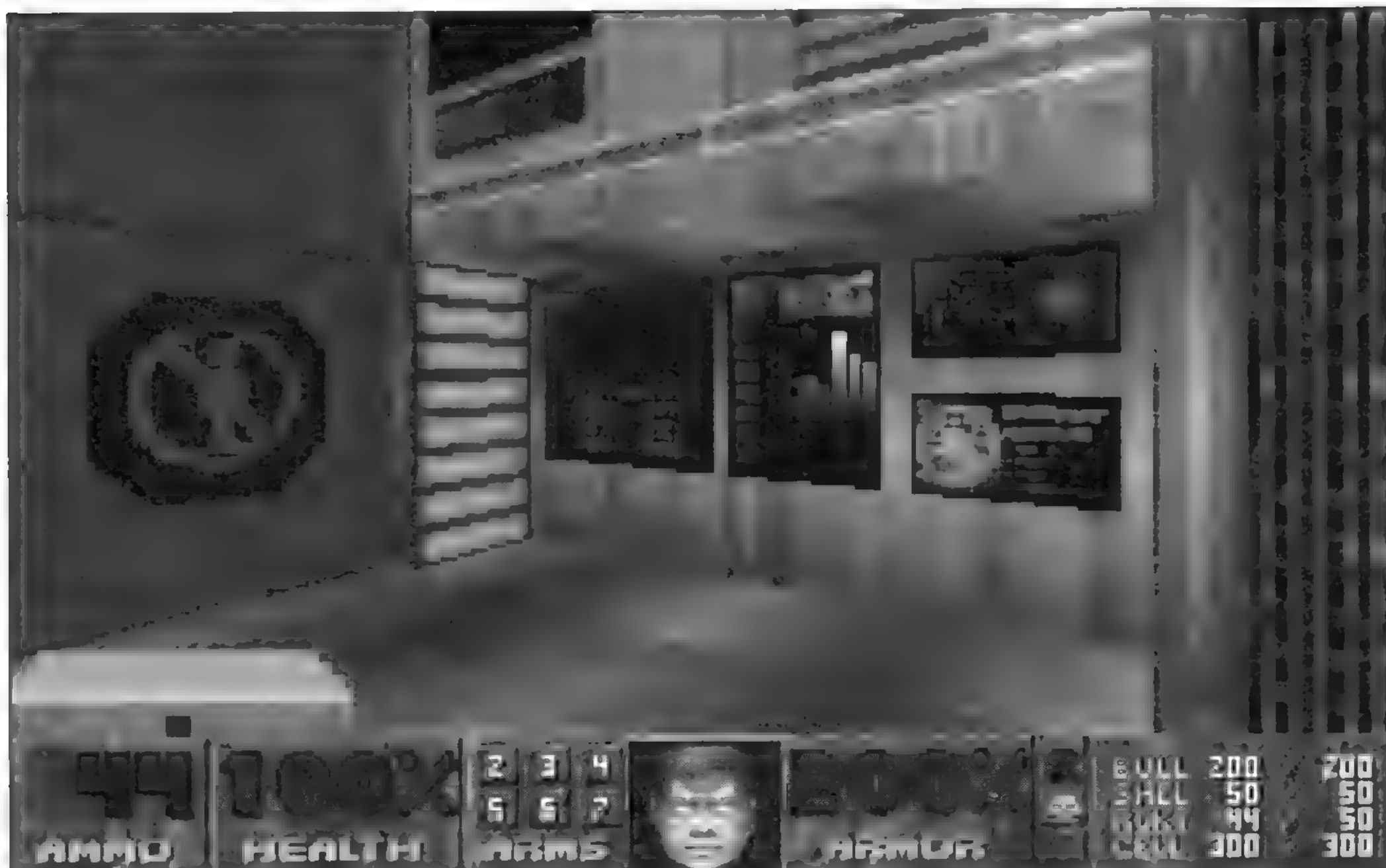


FIGURE 13.7: This is how the small computer room appears in DOOM.



FIGURE 13.8: The same computer room area in *Duke Nukem 3D* shows a darker sector under the panel. This sector is flashing because of a Cycler sprite.

up some Cycler sprites in the appropriate sectors. Figures 13.7 and 13.8 show this area in each game.

THE WATER VATS

The room beyond the computer areas is a large room with a strange alien texture on the walls. Three long pools of green slime run parallel throughout the room, as you can see in Figure 13.9. Each pool has a platform jutting out of it, and the intention of the original *DOOM* level designer was to have the player lower these platforms with a switch so they could be jumped on. Once again, the ability of the *Duke Nukem 3D* player to jump or fly defeats the purpose of this obstacle.

Instead of harmful green slime, I changed the long sectors to water-based sectors and added corresponding underwater sectors. This allows the player to jump in and explore underneath the vats. This is a good example of adding a feature to a converted level that couldn't exist in the original *DOOM* level. To make things a bit more interesting, I also added an underwater tunnel that connected to a pool in the control room, which will be described in the next section. Figure 13.10 shows the water (slime) vats after conversion to a *Duke Nukem 3D* level.



FIGURE 13.9: These slime vats in the *DOOM* level will be converted to water vats.

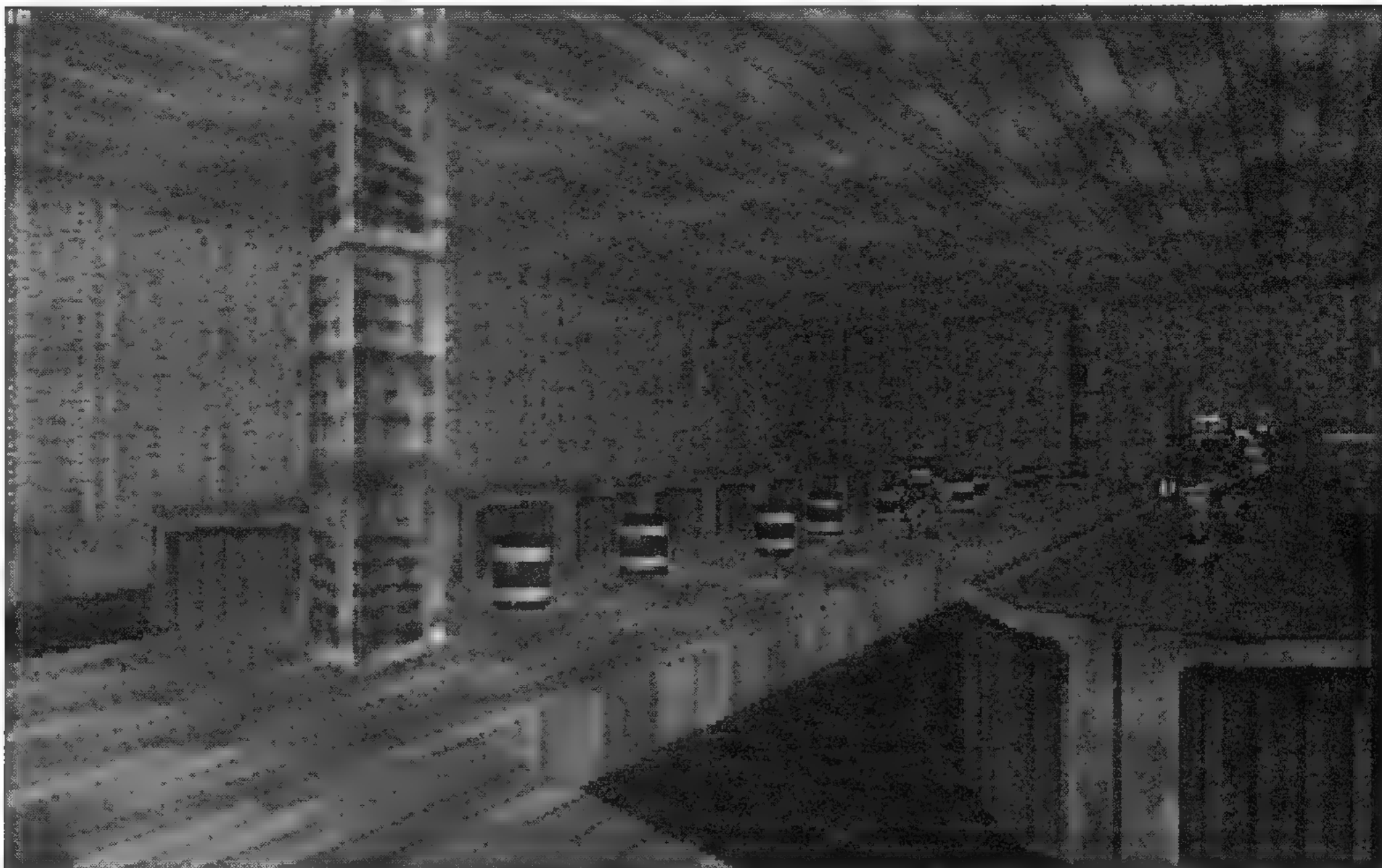


FIGURE 13.10: In the converted *Duke Nukem 3D* level the slime vats are now water vats with new underwater sectors.

THE CONTROL ROOM

This room lies on the far side of the water vats. I added a small pool of water to allow the player an alternate means of entrance and exit from this room via an underwater exit that connected to one of the water vats. In addition, I thought the northern wall of this room was a good place to put an exploding wall. The trigger for the wall is a fire extinguisher. Figures 13.11 and 13.12 show this area in both games.

COMPLETING THE CONVERSION

The small demonstration discussed in this chapter shows how a *DOOM* level can be transformed into a fabulous *Duke Nukem 3D* level using WAD2DUKE. Of course, the translation is not fully automated because many features found in *Duke Nukem 3D* simply don't exist in *DOOM*. Therefore, to take full advantage of the *Duke Nukem 3D* features, additional designing is necessary after the initial conversion to make the old *DOOM* level more thrilling in *Duke Nukem 3D*.



FIGURE 13.11: This is how the control room appears in *DOOM*.

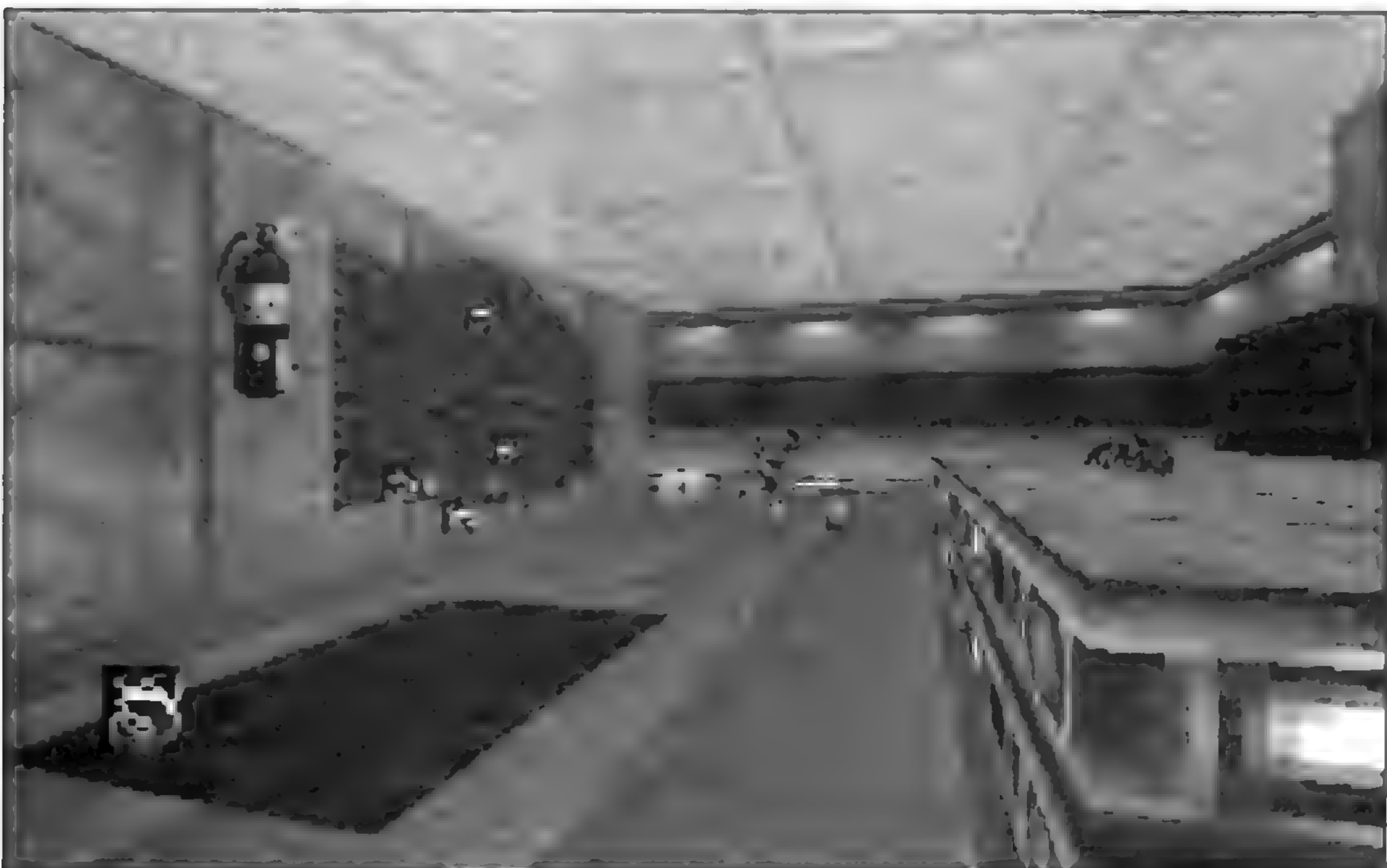


FIGURE 13.12: The converted control room includes an exploding wall and a pool.





Appendices

APPENDIX A

Making Memorable Levels

While reading the chapters, you probably came up with a few good ideas for a level of your own. But knowing how to use Build is only the first part of making a level; you also need to know what really makes a *good* level and what types of elements differentiate a good level from a mediocre one.

There are two people who have used Build more than anyone to create *Duke Nukem 3D* levels: Richard Bailey Gray (aka the Levelord) and Allen H. Blum III. They are the designers of all the levels in the retail version of *Duke Nukem 3D*. (The file `DUKELVLS.TXT`, which resides on the CD-ROM included with this book, lists each level and its designer.) Here, these gentlemen offer their own insights to explain the difference between a good level and a bad one. They are, perhaps, the most qualified to do this because they have spent the most time with Build and created the most levels.

THE LEVELORD'S 17 DESIGN POINTS

Level designing is a juxtaposition of art and engineering. Levels must be interesting to the senses and emotions, and they must also follow certain rules and regulations. A good level designer will know, first, what will make a cool level. That's the artistic part. A greater level designer will know the tools intimately, the ways to push the envelope, and the limitations that must be observed. The following are my 17 points of level design.

1. THE MAIN THEME

The most fundamental component of a cool level, and the first thing you must develop, is its theme. It's usually simple to find a cool theme. I get a lot of my ideas from movies, television, and illustrated books. Those comic books and fantasy/science-fi magazines that you were told were nothing but mere rubbish are also excellent sources. Don't be afraid to *appropriate* ideas, either. There's nothing wrong with borrowing, as long as it isn't outright stealing. When your level is done, there will be much more of you in it than the original inspiration, and you will be rightfully proud. Watch documentaries about ancient civilizations. Those civilizations of yore lasted eons, and their success is often manifested in their architectures. If you study their techniques, you'll find many neat things to use in your levels.

2. FRAME RATE

The next component of a good level is the one that separates the professionals from the amateurs—frame rate. Performance is the bottom line and should, actually, be design point 1. If a level runs slowly, usually because complex scenes drag the game engine to a halt, then the level is a complete failure. Anyone can think of cool themes and scenes. A good level designer, though, will pay absolute attention to frame rate in every possible view of every scene in the level, and a good level designer will not hesitate to kill the coolest of ideas for the sake of good frame rate. This is often heart-breaking, but it must be done.

Maintaining good frame rates throughout your level means that you must learn the game's engine intimately. You must know what will slow it down; the usual offenders in Duke's case are too many ornamental sprites and overly intricate scenes. Manifest your cool ideas in the simplest way possible and, again, kill them if you just can't get them to run fast.

3. CONTINUITY

Continuity is the third most important factor of a good level. Continuity is related to the level's main theme—the main theme must be maintained throughout the entire level. Too often I see levels that are patchworks of various themes. Your level must seem like a continuous place. After deciding on a theme, pick a tile set (a few wall textures and matching floors and ceilings) that fit it. Stay with that tile set throughout the level.

4. DETAILS AND REALISM

Think of every detail as you develop your level. The devil lives in details, and so does the player's eye. Pay close attention to every detail and what it takes to make your level as real as possible. The two most basic details are shadowing and appropriate texturing. This sounds easy enough, but many levels suffer from blandness because the designers did not take the time to cast shadows, or they agonize in dizzying confusion because the wrong textures were used. Real-world situations usually make good levels because it's easier to capture the reality. The closer the player can relate to your level, the deeper the player will be immersed into the level. Don't forget frame rate, though!

5. APPROPRIATE ENEMIES AND WEAPONS

Make sure that enemies fit the areas of your level. Try, also, to match the weapons and goodies that you provide the player to each area and its denizens.

6. THE RULE OF THE BRUCES

There is no design point 6.

7. THE STICK AND THE CARROT

A good level should be a series of challenges and rewards. The challenge can come before the reward or after it. Don't just strew goodies and bad guys through your level haphazardly. Players should feel as though they are being run through a gauntlet of contests and prizes. Remember design point 5: Make your challenges and rewards relate to each other, as well as to the main theme. If the challenge is a particular bad guy and the reward is a cool weapon, make the weapon be the one best used against that bad guy. If the challenge is an environmental hazard (lava), make the reward whatever utility the player needs to survive it (boots).

8. PICK YOUR AUDIENCE AND BUILD AROUND IT

You must decide whether your level is to be played by one person or many. Single-player levels can make for good DukeMatch levels, and vice versa, but often the two strategies conflict. Keep this in mind when you're designing the main layout of your

level. A good single-player level will have a critical path that leads the player through the level in a linear manner. The critical path intentionally denies access to areas of the level until the player has earned it. The critical path is just like a story line—it needs to start out slow and quiet, then build to a crescendo, and finally deliver the player to the exit.

A good single-player level also tends to be big and contains many areas to explore. You want your single-player audience to have lots of fun hunting and searching. You will have more puzzles and obstacles in a single-player level. A good multiplayer level, however, needs to be open and small so that the players can find each other quickly. Any two players must be able to find one another within a minute or two, or your level will not be played. Puzzle-solving and environmental hazards are usually found less often in multiplayer levels.

Sometimes, the two types of audiences can be appeased by making the critical path loop-shaped. Access can be denied to the single player via keycards (immediately available to all players in a multiplayer level), and areas can be opened up with multiplayer-only blowouts. You can also use multiplayer-only switches and teleporters.

9. PLAYER INTERACTION

Provide players with as much interaction stuff as possible. Players love blowing things up, shooting out lights, and blasting items made of glass. Take the time to become a good demolition expert. Make as many secret areas as possible, too. Put a hidden cubby behind that wall picture, and make that soda machine slide away to reveal something.

10. A WELL-BALANCED LEVEL

Make sure that your level is balanced for weapons and bad guys. In single-player levels, this means that the player should have just enough ammo and health to replenish what the bad guys are going to take away. Not enough goodies will mean you have a level that can't be finished; excessive goodies will mean that the level is too easy and thus boring. This can be hard when considering the different skill levels, so be patient. Remember that your audience will not be as familiar with your level (where each bad guy is, etc.), so provide a little extra of the goodies than you yourself would need.

Balancing is easier to accomplish for multiplayer levels. You only need to ensure that weapons and goodies are scattered enough so that all players can stock themselves readily. Usually multiplayer levels will have extra goodies not found in the single-player

version. Your layout of multiplayer goodies should induce the players to navigate your intended loop.

11. PUSH THE ENVELOPE

Almost in contrast to design point 4 (realism), it is good to find the bounds of the game engine and push them as far as you can. The game is only virtual reality, and you can amaze the player with some unreal entertainment. If you've seen the Lunatic Fringe and Tier Drop levels, you'll know what I mean about breaking reality.

12. PATTERNS

Humans love patterns. Recognizing patterns and fitting our behavior to them is what our minds crave, and any entertainment will use this hunger to engage an audience. A good level usually has a basic geometric pattern to it. This is pleasing to the spatial part of our brain, and it helps the player solve the general level. Also, try to think in beats when you place your goodies and bad guys along the critical path—for example, trooper, trooper, commander, reward, trooper, trooper, commander, reward. Don't be too monotonous, though.

13. CONNECTIONS

Humans also love connections, linking two segments into one. When you can, provide a view of areas yet to come. There's something wonderfully pleasing about seeing an area early in the level and then gaining access to it later.

14. PUZZLES

It's not just killing bad guys that makes 3D immersion games fun. You should also include as much puzzle-solving in your levels as possible. Good puzzles are usually in the form of mazes, hip-hops, and obstacle-course goodies. Don't make your puzzles so hard, though, that they only frustrate the player. Often, you will be able to solve a puzzle of your own creation because you know all its tricks and secrets, but an unknowing player will not be so educated and will not want to spend too much time figuring it out.

If your level is strictly for more than one player, you will want to keep the puzzles and any time drains to a minimum. Puzzles are mostly for single play, although they

can be used in multiplayer levels if they also provide good opportunities for sniper attacks. This is done by making sure other players can attack the player who's trying to get at the cool reward at the end of the puzzle.

15. THINK SNEAKY

Make the player work for the rewards. Do whatever you can to make the player say, "Ah ha!" Try to fool the player into thinking there is an obvious way to tackle an obstacle, but make this apparent answer unusable. Then sneak in a hidden solution. Secret messages, ones seen only with night vision goggles, are great for revealing the secret to success. Again, do not frustrate the player too much.

16. THE FIDDLER'S ROOF

Almost in contrast to design point 8, don't always force the single player to stay on the level's critical path. Sometimes let the players go where they're not supposed to, and then demolish them with bad guys or hazards that they're not equipped for yet.

17. PRACTICE WHAT YOU PREACH

The last but definitely not the least good level-design point is play-testing. Play-test your level until you are sure you've checked everything. Play-test it some more, and then play-test it again. You should be completely sick of your level before you consider it finished. No matter what the target audience of your map will be—single players, multiple players, or whoever—play-test it under those exact conditions.

BLUM'S DUKE MAP CHECKLIST

After you come up with a cool setting and map out the general layout, you need to check six basic points to finish a map: frame rate, single-player levels, multiplayer levels, sounds, secrets, and skill levels. If you are making a map for DukeMatch only, you can skip the single-player and skill level points, and maybe secrets, but frame rate and sounds are very important in DukeMatch for many reasons. When you've checked these points, your map should be great for anyone to play.

FRAME RATE

For the designers of *Duke Nukem 3D*, frame rate was the most important consideration in making a map playable on most machines. Our base machine is a 486/66 at 20 frames per second in Build. This would mean that during game play, the frame rate would be between 15 frames per second (fps) and 20 fps, depending on how much action was happening.

After completing the basic layout of the level, I would go in the corner of each room and area in the map and look for slow frame rates. If there was ever a rate less than 20 fps, then I would make modifications to that area until it became at least 20 fps. One way to increase frame rate is to minimize how much of an area you can see. For example, block off half the scene with a turned-over semitrailer or by putting a large building in the way. Another great way is to take out some of the unneeded details that are for looks only or that have no game play value. For example, if twenty palm trees line a road, get rid of at least half of them, and it will have the same look but at less cost in frame rate. Even with one palm tree, players still get the feeling of trees in the area because their minds fill in the blanks as they are dealing with the action of the game. All these little sacrifices will pay off in the end when you are playing DukeMatch, your aiming is smooth with the mouse, and you never miss shooting your best friend.

SINGLE-PLAYER LEVELS

The first thing to think about is a path of destruction—what route the player needs to take to finish the map. I always try to start with an area that looks very cool but has few to no bad guys. This gives players the chance to get used to the new setting and gets them excited about going forward in the map. Then through the rest of the map, I don't need to detail as much because the player already has a good mental image of the area and I can achieve a good frame rate.

When placing the bad guys, always make it easy at the beginning of the map, and slowly make it harder toward the exit. I usually make it easy at the beginning to build up the players' confidence, and then destroy them by the end. Don't forget to put bad guys in places where they can jump out and catch players off guard. This will keep them scared enough so they won't want to blind their eyes and enjoy the map. As for weapons, I always prefer the players to come to the map with nothing. So as they play the map, I slowly give them new weapons, usually starting with the shotgun and working my way up to the stronger weapons. When you put in ammo for the weapons, always make sure you have first given them the weapon with which to use the ammo.

One last thing to make sure people can play through the map is to play-test it. Because I know where everything is in my maps, I try to play very sloppily to see if there is enough ammo and health to finish the level. This should enable most people to play the map. Later I will add the skill levels to take the map from being easy to being hard.

MULTIPLAYER LEVELS

In multiplayer levels, the main thing to remember is to have at least one of each weapon, ammo, and item in the map. To see how many of each object is in the map, in Build's 2D view mode press the F5 key. The numbers in cyan are single-player items, and the numbers in dark blue are multiplayer items only. This helps to ensure a map is not overpopulated with health and ammo.

When placing the weapons around the map, I try to put them in locations that can be reached from multiple directions. This will make it easier for more players to be able to get to the weapons, so that one person can't defend and always have a particular weapon. Also, try to put the ammo for that weapon somewhere else in the map so the player has to move around the map to keep fighting.

As for health placement, I try to place them out in the open so players have to come out of their hiding places to get healthy. The best thing is to have atomic health units in the center of a large area to draw the action out in the open. To keep the action flowing, it's always good to localize the placement of items, and make sure you can get to any place in the map in less than seven seconds. A good way to do this is to have secret passageways from place to place that don't affect the main path in single-player levels. Usually, I make sure the secrets are found late in the map and lead back to areas where the player has already been. One of the most important things to remember in multiplayer levels is the use of sound effects. If the other players activate a door or use some sort of device, then when you hear it active, you know just where to go to get them.

SOUNDS

Sounds are a very important method of making the player feel that he or she is somewhere real. I always make sure there are environment sounds around areas that visually look active, such as around waterfalls or moving water, where there would be loud water sounds. In the streets there are sounds of activity, such as jets flying by, screams, and explosions. One last check to a level I always make is to walk through it with no

bad guys active to see if everything that should make a sound is making a sound, such as doors, the activation of earthquakes, and special effects. A lot of the mood in a map is conveyed through the use of sounds.

SECRETS

For single-player levels, there is nothing more rewarding than finding a secret place. Some of these places might just be a hidden door that opens to a room with a new weapon or something more obscure, like a crack in a wall that when blown out leads to a secret area that has a hidden message. Another type of secret could lead the player to a good position to get the drop on a slew of bad guys or a great sniper position for multiplayer situations. This is a great way to make people replay a map over and over again just to look for the crazy secrets that can be found.

SKILL LEVELS

The last point in my checklist is to make the map is playable by everybody across all skill levels. When I first put the bad guys around the map, I make it playable on the default skill level. Then I go back through the map and try to thin out areas that are thickly populated. For example, if a room has three bad guys, I would tag it so that there would be one for easy mode and three for the default mode. For the hard skill mode, I might add one or two more bad guys or just one really tough bad guy. Then I would play through all the different skill modes to see if it was still playable. Assume that the hard skill mode should be very difficult for most players. The hardest skill mode should be almost impossible. This made it tough for me, because I had to be able to play through all skill modes with no cheats active. But I knew it was right when I started to sweat while playing the hardest skill mode.

After checking off all these points in my checklist, I could assume the level was done. Then it was time to have others play through it one last time to find any problems. If they came back with a look of stress on their face, then I knew that they had had fun.

APPENDIX B

Sounds List

The following is a complete list of all the sounds available in *Duke Nukem 3D*. I am not the one to be given credit for the exhaustive work of compiling this list. The effort was made by Shane King, also known as the Scatt Man. Shane graciously allowed me to include his list in the book, for handy reference by you, the Build designer. The list appears exactly as is did upon Shane's upload, without alteration.

COMPLETE LIST OF *DUKE NUKEM 3D* SOUNDS

By Shane King aka Scatt Man (scattman@bssc.edu.au)

FORMAT OF THIS LIST

* sound number

Used in build

* sound name

This is the name that Duke Nukem uses internally to reference the sound

* .voc file name

The .voc files are stored in the duke3d.grp file

* min frequency adjustment

* max frequency adjustment

A random value between these two limits is chosen each time the sound is played

* priority

If there are too many sounds to be played at once the higher (I think) priority sounds get played

* attributes

- bit 0 (%) Repeat
This sound will be played over and over
- bit 1 (\$) Ambient Sound
This sound can be used as an ambient sound
- bit 2 (#) Duke Talk
This sound will follow the player
- bit 3 (@) Adult Mode
This sound will only be played if adult mode is on
- bit 4 (!) Random Ambient Sound
This sound can be used as a random ambient sound

* volume adjustment

Used to make the sound louder or softer

```
* sample rate (Khz)
```

Multi means the .voc file has multiple blocks

```
* description of sound
```

A + in any section denotes that the section is not defined, hence following sections are also not defined

| | Event | Weapon | Cost | Value | Weight | Count | Drop Rate | Notes |
|---|---------------------------------|----------|-------|-------|--------|-----------|-----------|--------|
| + | SLIM_PAIN | slimprn | 0 | 0 | 3 | - - - - - | 0 | + |
| 0 | KICK_HIT | kickhit | 0 | 0 | 4 | - - - - - | 0 | 5.988 |
| | Duke's Mighty foot hits | | | | | | | |
| 1 | PISTOL_RICOCHET | ricochet | 0 | 0 | 0 | - - - - - | 4096 | 5.988 |
| | Pistol ricochet of solid object | | | | | | | |
| 2 | PISTOL_BODYHIT | bulithit | 0 | 0 | 0 | - - - - - | 0 | 5.988 |
| | A bullit hits somebody | | | | | | | |
| 3 | PISTOL_FIRE | pistol | -64 | 0 | 254 | - - - - - | 0 | 5.988 |
| | Pistol firing | | | | | | | |
| 4 | EJECT_CLIP | clipout | 0 | 0 | 3 | - - - - - | 0 | 5.988 |
| | Duke ejects a used clip | | | | | | | |
| 5 | INSERT_CLIP | clipin | 512 | 512 | 3 | - - - - - | 0 | 5.988 |
| | Duke inserts a new clip | | | | | | | |
| 6 | CHAINGUN_FIRE | chaingun | -204 | -204 | 254 | - - - - - | 512 | 10.989 |
| | Chaingun firing | | | | | | | |
| 7 | RPG_SHOOT | rpgfire | -32 | 0 | 4 | - - - - - | 0 | 5.988 |
| | RPG Firing | | | | | | | |
| 8 | POOLBALLHIT | poolball | 0 | 0 | 0 | - - - - - | 0 | 8 |
| | A poolball is hit | | | | | | | |
| 9 | RPG_EXPLODE | bombexpl | -1024 | 1024 | 254 | - - - - - | 0 | multi |
| | A RPG rocket explodes | | | | | | | |

| | | | | | | | | | | | | |
|----|--|----------|------|-----|-----|---|---|---|---|---|------|--------|
| 10 | CAT_FIRE | catfire | 512 | 768 | 4 | - | - | - | - | - | 0 | 5.988 |
| | Freeze Ray Firing | | | | | | | | | | | |
| 11 | SHRINKER_FIRE | shrinker | -512 | 0 | 4 | - | - | - | - | - | 0 | 8 |
| | Shrinker Ray Firing | | | | | | | | | | | |
| 12 | ACTOR_SHRINKING | shrink | 0 | 0 | 2 | - | - | - | - | - | 0 | 8 |
| | Somebody being shrunk | | | | | | | | | | | |
| 13 | PIPEBOMB_BOUNCE | pbombbnc | 0 | 0 | 2 | - | - | - | - | - | 6144 | 5.988 |
| | Tink of a pipe bomb bouncing | | | | | | | | | | | |
| 14 | PIPEBOMB_EXPLODE | bombexpl | -512 | 0 | 128 | - | - | - | - | - | 0 | multi |
| | A pipe bomb being detonated | | | | | | | | | | | |
| 15 | LASERTRIP_ONWALL | lsrbmbpt | 0 | 0 | 3 | - | - | - | - | - | 0 | 5.988 |
| | A laser trip bomb being placed on a wall | | | | | | | | | | | |
| 16 | LASERTRIP_ARMING | lsrbmbwn | 0 | 0 | 3 | - | - | - | - | - | 0 | 5.988 |
| | Beeeping of a laser trip bomb about to explode | | | | | | | | | | | |
| 17 | LASERTRIP_EXPLODE | bombexpl | -512 | 0 | 4 | - | - | - | - | - | 0 | multi |
| | A laser trip bomb exploding | | | | | | | | | | | |
| 18 | VENT_BUST | ventbust | -32 | 32 | 2 | - | - | - | - | - | 0 | 5.988 |
| | Breaking a vent or fan | | | | | | | | | | | |
| 19 | GLASS_BREAKING | glass | -412 | 0 | 3 | - | - | - | - | - | 8192 | 8 |
| | Glass window being smashed | | | | | | | | | | | |
| 20 | GLASS_HEAVYBREAK | glashevy | -412 | 0 | 3 | - | - | - | - | - | 8192 | 11.025 |
| | Glass items being broken | | | | | | | | | | | |
| 21 | SHORT_CIRCUIT | shorted | 0 | 0 | 0 | - | - | - | - | - | 6500 | 8 |
| | Duke gets an electric shock | | | | | | | | | | | |
| 22 | ITEM_SPLASH | splash | 0 | 0 | 2 | - | - | - | - | - | 0 | multi |
| | Item dropped into water | | | | | | | | | | | |
| 23 | DUKE_BREATHING | hlminhal | 0 | 0 | 255 | - | - | # | - | - | 0 | + |
| 24 | DUKE_EXHALING | hlmexhal | 0 | 0 | 255 | - | - | # | - | - | 0 | + |
| 25 | DUKE_GASP | gasp | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 |
| | Duke catching breath after being under water | | | | | | | | | | | |
| 26 | SLIM_RECOG | slirec06 | 0 | 0 | 3 | - | - | - | - | - | 0 | 11.025 |
| | Green slime sights Duke | | | | | | | | | | | |
| 27 | ENDSEQVOL3SND1 | KICKHEAD | 0 | 0 | 254 | - | - | - | - | - | 0 | 11.025 |
| | Duke kicks boss's head at goal | | | | | | | | | | | |
| 28 | DUKE_URINATE | pissing | 0 | 0 | 4 | - | - | - | - | - | 0 | 5.988 |
| | Duke going to the toilet | | | | | | | | | | | |
| 29 | ENDSEQVOL3SND2 | GMEOVR05 | 0 | 0 | 254 | - | - | - | - | - | 0 | 8 |
| | Duke "Game over" | | | | | | | | | | | |
| 30 | ENDSEQVOL3SND3 | CHEER | 0 | 0 | 254 | - | - | - | - | - | 0 | 11.025 |
| | The crowd cheering Duke | | | | | | | | | | | |

APPENDIX B Sounds List

[illegible]

| Line | Event | Sound | Start | End | Volume | Frequency | Duration | Notes |
|------|--|----------|-------|-----|--------|------------|----------|--------|
| 55 | LIZTROOP_TALK3 | + | | | | | | |
| 56 | DUKETALKTOBOSS | duknu14 | 0 | 0 | 255 | - @ # - - | 0 | 8 |
| | Duke "I'm Duke Nukem and I'm coming to get the rest of you alien bastards" | | | | | | | |
| 57 | LIZCAPT_GROWL | + | | | | | | |
| 58 | LIZCAPT_TALK1 | + | | | | | | |
| 59 | LIZCAPT_TALK2 | + | | | | | | |
| 60 | LIZCAPT_TALK3 | + | | | | | | |
| 61 | LIZARD_BEG | chokn12 | 0 | 0 | 3 | - - - - - | 0 | multi |
| | Alien shaking head when not quite dead | | | | | | | |
| 62 | LIZARD_PAIN | + | | | | | | |
| 63 | LIZARD_DEATH | + | | | | | | |
| 64 | LIZARD_SPIT | lizspit | 0 | 0 | 0 | - - - - - | 0 | 8 |
| | Lizard spitting at Duke | | | | | | | |
| 65 | DRONE1_HISSRATTLE | + | | | | | | |
| 66 | DRONE1_HISSSCREECH | + | | | | | | |
| 67 | DUKE_TIP2 | shake2a | 0 | 0 | 255 | - - # - - | 0 | 8 |
| | Duke "Shake it baby" | | | | | | | |
| 68 | FLESH_BURNING | fire09 | -256 | 0 | 0 | - - - - - | 6100 | multi |
| | Fire crackling | | | | | | | |
| 69 | SQUISHED | squish | 0 | 0 | 3 | - - - - - | 0 | 8 |
| | Someone is squished | | | | | | | |
| 70 | TELEPORTER | teleport | 0 | 0 | 0 | - - - - - | 0 | 5.988 |
| | A teleporter is used | | | | | | | |
| 71 | ELEVATOR_ON | gbelev01 | 0 | 0 | 0 | - - - - - | 0 | 8 |
| | Elevator is used | | | | | | | |
| 72 | DUKE_KILLED3 | thsuk13a | 0 | 0 | 255 | - @ # - - | 0 | 8 |
| | Duke "Uugh, this sucks" | | | | | | | |
| 73 | ELEVATOR_OFF | gbelev02 | 0 | 0 | 0 | - - - - - | 0 | 8 |
| | Elevator stopping | | | | | | | |
| 74 | DOOR_OPERATE1 | slidoor | -256 | 0 | 0 | - - - - - | 0 | 11.025 |
| | Metalic Sliding Door | | | | | | | |
| 75 | SUBWAY | subway | 0 | 0 | 0 | - - - - - | 0 | multi |
| | A subway train | | | | | | | |
| 76 | SWITCH_ON | switch | 0 | 0 | 0 | - - - - - | 0 | multi |
| | Click of a switch being used | | | | | | | |
| 77 | FAN | fan | 0 | 0 | 0 | - - - - - | 0 | + |
| 78 | DUKE_GETWEAPON3 | groovy02 | 0 | 0 | 255 | - - # - - | 0 | 8 |
| | Duke "Groovy" | | | | | | | |
| 79 | FLUSH_TOILET | flush | 0 | 0 | 3 | - - - \$ - | 0 | multi |
| | Toilet being flushed | | | | | | | |

| | | | | | | | | | | | | |
|-----|--|----------|------|---|-----|---|---|---|----|---|------|--------|
| 80 | HOVER_CRAFT | hover | 0 | 0 | 0 | - | - | - | - | - | 0 | + |
| 81 | EARTHQUAKE | quake | 0 | 0 | 0 | - | - | - | - | - | 0 | multi |
| | Earthquake rumble | | | | | | | | | | | |
| 82 | INTRUDER_ALERT | alert | 0 | 0 | 0 | - | - | - | - | - | 0 | 5.988 |
| | Warning alarm | | | | | | | | | | | |
| 83 | END_OF_LEVEL_WARN | monitor | 0 | 0 | 0 | - | - | - | - | - | 0 | 8 |
| | End of level warning | | | | | | | | | | | |
| 84 | ENGINE_OPERATING | onboard | 0 | 0 | 0 | - | - | - | \$ | - | 0 | multi |
| | Low pitched grind of engine operation | | | | | | | | | | | |
| 85 | REACTOR_ON | reactor | 0 | 0 | 0 | - | - | - | \$ | - | 0 | multi |
| | High pitched hum of reactor | | | | | | | | | | | |
| 86 | COMPUTER_AMBIENCE | compamb | 0 | 0 | 0 | - | - | - | \$ | - | 0 | 11.025 |
| | Computer operation sound | | | | | | | | | | | |
| 87 | GEARS_GRINDING | geargrnd | 0 | 0 | 0 | - | - | - | \$ | - | 0 | 5.988 |
| | Turning gears grinding together | | | | | | | | | | | |
| 88 | BUBBLE_AMBIENCE | bubblamb | -256 | 0 | 0 | - | - | - | \$ | - | 0 | multi |
| | Water bubbling | | | | | | | | | | | |
| 89 | MACHINE_AMBIENCE | machamb | 0 | 0 | 0 | - | - | - | \$ | - | 0 | 11.025 |
| | Thumping of machine operating | | | | | | | | | | | |
| 90 | SEWER_AMBIENCE | drip3 | 0 | 0 | 0 | - | - | - | - | - | 0 | 11.025 |
| | Drip into water | | | | | | | | | | | |
| 91 | WIND_AMBIENCE | wind54 | 0 | 0 | 0 | - | - | - | \$ | - | 0 | multi |
| | Gust of wind | | | | | | | | | | | |
| 92 | SOMETHING_DRIPPING | drip3 | 0 | 0 | 0 | - | - | - | - | - | 9000 | 11.025 |
| | Loud drip | | | | | | | | | | | |
| 93 | STEAM_HISSING | steamhis | 0 | 0 | 0 | - | - | - | - | % | 8192 | 8 |
| | Steam hissing from pipe or pot | | | | | | | | | | | |
| 94 | THEATER_BREATH | + | | | | | | | | | | |
| 95 | BAR_MUSIC | barmusic | 0 | 0 | 254 | - | - | - | \$ | % | 0 | 11.025 |
| | Dance music in bar (as in bar in e112) | | | | | | | | | | | |
| 96 | BOS1_ROAM | bos1rm | 0 | 0 | 3 | - | - | - | - | - | 0 | 11.025 |
| | Boss 1 screaming | | | | | | | | | | | |
| 97 | BOS1_RECOG | bos1rg | 0 | 0 | 5 | - | - | - | - | - | 0 | 8 |
| | Boss 1 sees Duke | | | | | | | | | | | |
| 98 | BOS1_ATTACK1 | chaingun | 0 | 0 | 3 | - | - | - | - | - | 0 | 10.989 |
| | Boss 1 using chaingun | | | | | | | | | | | |
| 99 | BOS1_PAIN | bos1pn | 0 | 0 | 3 | - | - | - | - | - | 0 | 8 |
| | Boss 1 in pain | | | | | | | | | | | |
| 100 | BOS1_DYING | bos1dy | 0 | 0 | 3 | - | - | - | - | - | 0 | 8 |
| | Boss 1 biting the dust | | | | | | | | | | | |

| | | | | | | | | | | | | |
|-----|---|----------|------|-----|---|---|---|---|---|---|------|--------------|
| 121 | PIG_RECOG | pigrgr | -200 | 400 | 3 | - | - | - | - | - | 0 | 8 |
| | Pig Cop sees Duke | | | | | | | | | | | |
| 122 | PIG_ATTACK | shotgun7 | -256 | 256 | 4 | - | - | - | - | - | 0 | 11.025 16bit |
| | Pig Cop fires shotgun | | | | | | | | | | | |
| 123 | PIG_PAIN | pigpn | 100 | 800 | 3 | - | - | - | - | - | 0 | 8 |
| | Pig Cop grunts in pain | | | | | | | | | | | |
| 124 | PIG_DYING | pigdy | -800 | 100 | 3 | - | - | - | - | - | 0 | 8 |
| | Pig Cop death squeal | | | | | | | | | | | |
| 125 | RECO_ROAM | jetpaki | 0 | 0 | 3 | - | - | - | - | - | 0 | 8 |
| | Recon Car Moving | | | | | | | | | | | |
| 126 | RECO_RECOG | pigrgr | 0 | 0 | 3 | - | - | - | - | - | 0 | 8 |
| | Recon Car sees Duke | | | | | | | | | | | |
| 127 | RECO_ATTACK | gblasr01 | 256 | 256 | 3 | - | - | - | - | - | 7680 | 8 |
| | Recon Car firing at Duke | | | | | | | | | | | |
| 128 | RECO_PAIN | pigpn | 0 | 0 | 3 | - | - | - | - | - | 0 | 8 |
| | Recon car grunts in pain | | | | | | | | | | | |
| 129 | RECO_DYING | pigdy | 0 | 0 | 3 | - | - | - | - | - | 0 | 8 |
| | Recon Car Dies | | | | | | | | | | | |
| 130 | DRON_ROAM | snakrm | 0 | 0 | 3 | - | - | - | - | - | 0 | multi |
| | Drone roaming | | | | | | | | | | | |
| 131 | DRON_RECOG | snakrg | 0 | 0 | 3 | - | - | - | - | - | 0 | 11.025 |
| | Drone sees Duke | | | | | | | | | | | |
| 132 | DRON_ATTACK1 | snakata | 0 | 0 | 3 | - | - | - | - | - | 0 | 11.025 |
| | High pitched zzzzz when drone attacks Duke | | | | | | | | | | | |
| 133 | DRON_PAIN | snakpn | 0 | 0 | 3 | - | - | - | - | - | 0 | 11.025 |
| | Drone in Pain | | | | | | | | | | | |
| 134 | DRON_DYING | snakdy | 0 | 0 | 3 | - | - | - | - | - | 0 | 11.025 |
| | Growl when Drone dies | | | | | | | | | | | |
| 135 | COMM_ROAM | commrm | 0 | 0 | 3 | - | - | - | - | - | 0 | 11.025 |
| | Fat commander roaming | | | | | | | | | | | |
| 136 | COMM_RECOG | commrg | 0 | 0 | 3 | - | - | - | - | - | 0 | 11.025 |
| | Fat Commander sees Duke and says "Die Human" | | | | | | | | | | | |
| 137 | COMM_ATTACK | commat | 0 | 0 | 3 | - | - | - | - | - | 0 | 11.025 |
| | Fat Commander attacking and says "Suck it down" | | | | | | | | | | | |
| 138 | COMM_PAIN | commpn | 0 | 0 | 3 | - | - | - | - | - | 0 | 11.025 |
| | Fat Commander in pain | | | | | | | | | | | |
| 139 | COMM_DYING | commdy | 0 | 0 | 3 | - | - | - | - | - | 0 | 11.025 |
| | Extended scream of commander dying | | | | | | | | | | | |
| 140 | OCTA_ROAM | octarm | -200 | 0 | 3 | - | - | - | - | - | 0 | 8 |
| | Octobrain roaming | | | | | | | | | | | |

| | | | | | | | | | | | | |
|-----|--------------------------------|----------|------|------|---|---|---|---|---|---|---|--------|
| 141 | OCTA_RECOG | octarg | 0 | 0 | 3 | - | - | - | - | - | 0 | 8 |
| | Octobrain sees Duke | | | | | | | | | | | |
| 142 | OCTA_ATTACK1 | octaat1 | 0 | 0 | 3 | - | - | - | - | - | 0 | 8 |
| | Octobrain firing at Duke | | | | | | | | | | | |
| 143 | OCTA_PAIN | octapn | -400 | 0 | 3 | - | - | - | - | - | 0 | 8 |
| | Octobrain screaming in pain | | | | | | | | | | | |
| 144 | OCTA_DYING | octady | -400 | -100 | 3 | - | - | - | - | - | 0 | 8 |
| | Octobrain dying | | | | | | | | | | | |
| 145 | TURR_ROAM | turrrm | 0 | 0 | 3 | - | - | - | - | - | 0 | + |
| 146 | TURR_RECOG | turrrg | 0 | 0 | 3 | - | - | - | - | - | 0 | + |
| 147 | TURR_ATTACK | turrrat | 0 | 0 | 3 | - | - | - | - | - | 0 | + |
| 148 | DUMPSTER_MOVE | grind | 0 | 0 | 0 | - | - | - | - | - | 0 | 11.025 |
| | Grinding wheels of dumpster | | | | | | | | | | | |
| 149 | SLIM_DYING | slidie03 | 0 | 0 | 3 | - | - | - | - | - | 0 | 11.025 |
| | Green Slime dying | | | | | | | | | | | |
| 150 | BOS3_ROAM | b3roam01 | 0 | 0 | 3 | - | - | - | - | - | 0 | 11.025 |
| | Boss 3 roaming | | | | | | | | | | | |
| 151 | BOS3_RECOG | b3pain04 | 0 | 0 | 3 | - | - | - | - | - | 0 | 11.025 |
| | Boss 3 sees Duke | | | | | | | | | | | |
| 152 | BOS3_ATTACK1 | b3atk01 | 0 | 0 | 3 | - | - | - | - | - | 0 | + |
| 153 | BOS3_PAIN | b3rec03g | 0 | 0 | 3 | - | - | - | - | - | 0 | 11.025 |
| | Boss 3 in pain | | | | | | | | | | | |
| 154 | BOS1_ATTACK2 | rpgfire | 0 | 0 | 3 | - | - | - | - | - | 0 | + |
| 155 | COMM_SPIN | commsp | 0 | 0 | 3 | - | - | - | - | - | 0 | 11.025 |
| | Fat Commander spinning | | | | | | | | | | | |
| 156 | BOS1_WALK | thud | 0 | 0 | 3 | - | - | - | - | - | 0 | multi |
| | Thump as Boss 1 walks | | | | | | | | | | | |
| 157 | DRON_ATTACK2 | snakatB | 0 | 0 | 3 | - | - | - | - | - | 0 | 11.025 |
| | Drone attacking | | | | | | | | | | | |
| 158 | THUD | thud | 0 | 0 | 0 | - | - | - | - | - | 0 | multi |
| | THUD! | | | | | | | | | | | |
| 159 | OCTA_ATTACK2 | octaat2 | 0 | 600 | 3 | - | - | - | - | - | 0 | multi |
| | Octobrain attacking with teeth | | | | | | | | | | | |
| 160 | WIERDSHOT_FLY | octaat1 | 0 | 0 | 3 | - | - | - | - | - | 0 | 8 |
| | Octobrain firing | | | | | | | | | | | |
| 161 | TURR_PAIN | turrrpn | 0 | 0 | 3 | - | - | - | - | - | 0 | + |
| 162 | TURR_DYING | turrrdy | 0 | 0 | 3 | - | - | - | - | - | 0 | + |
| 163 | SLIM_ROAM | sliroa02 | 0 | 0 | 3 | - | - | - | - | - | 0 | 11.025 |
| | Green Slime roaming | | | | | | | | | | | |

| | | | | | | | | | | | | |
|-----|--|----------|------|-----|-----|---|---|---|----|---|---|--------|
| 164 | LADY_SCREAM | FSCRM10 | -256 | 0 | 254 | - | @ | - | - | - | 0 | 8 |
| | Woman screaming when hit | | | | | | | | | | | |
| 165 | DOOR_OPERATE2 | opendoor | -256 | 0 | 0 | - | - | - | - | - | 0 | 11.025 |
| | Swinging door | | | | | | | | | | | |
| 166 | DOOR_OPERATE3 | edoor10 | -256 | 0 | 0 | - | - | - | - | - | 0 | 11.025 |
| | Sliding door | | | | | | | | | | | |
| 167 | DOOR_OPERATE4 | edoor11 | -256 | 0 | 0 | - | - | - | - | - | 0 | 11.025 |
| | Sliding door | | | | | | | | | | | |
| 168 | BORNTOWILDsnd | 2bwild | 0 | 0 | 254 | - | - | - | \$ | - | 0 | 11.025 |
| | Born to be wild music | | | | | | | | | | | |
| 169 | SHOTGUN_COCK | shotgnck | 96 | 192 | 3 | - | - | - | - | - | 0 | 11.025 |
| | Shotgun being cocked | | | | | | | | | | | |
| 170 | GENERIC_AMBIENCE1 | grind | 0 | 0 | 0 | - | - | - | - | % | 0 | 11.025 |
| | Grinding Sound | | | | | | | | | | | |
| 171 | GENERIC_AMBIENCE2 | enghum | 0 | 0 | 0 | - | - | - | \$ | - | 0 | multi |
| | Engine humming | | | | | | | | | | | |
| 172 | GENERIC_AMBIENCE3 | lava06 | 0 | 0 | 0 | - | - | - | \$ | - | 0 | 8 |
| | Lava | | | | | | | | | | | |
| 173 | GENERIC_AMBIENCE4 | bubblamb | -256 | 0 | 0 | - | - | - | \$ | - | 0 | multi |
| | Water bubbling | | | | | | | | | | | |
| 174 | GENERIC_AMBIENCE5 | phonbusy | 0 | 0 | 0 | - | - | - | - | - | 0 | 11.025 |
| | Phone Engaged | | | | | | | | | | | |
| 175 | GENERIC_AMBIENCE6 | roam22 | 0 | 0 | 0 | - | - | - | \$ | - | 0 | multi |
| | Octobrain like sound | | | | | | | | | | | |
| 176 | BOS3_ATTACK2 | + | | | | | | | | | | |
| 177 | GENERIC_AMBIENCE17 | myself3a | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 |
| | Duke "Hmmm, don't have time to play with myself" | | | | | | | | | | | |
| 178 | GENERIC_AMBIENCE18 | monolith | 0 | 0 | 0 | - | - | - | \$ | - | 0 | 11.025 |
| | Weird alien ambience | | | | | | | | | | | |
| 179 | GENERIC_AMBIENCE19 | hydro50 | 0 | 0 | 0 | - | - | - | \$ | - | 0 | 11.025 |
| | Water ambience | | | | | | | | | | | |
| 180 | GENERIC_AMBIENCE20 | con03 | 0 | 0 | 0 | - | - | # | - | - | 0 | 11.025 |
| | Duke "Hmmm, looks like I have the con" | | | | | | | | | | | |
| 181 | GENERIC_AMBIENCE21 | !prison | 0 | 0 | 255 | - | - | # | - | - | 0 | multi |
| | "Ha Ha Ha, too late Nukem, we're in control now" | | | | | | | | | | | |
| 182 | GENERIC_AMBIENCE22 | vpiss2 | 0 | 0 | 255 | - | - | # | - | - | 0 | + |
| 183 | SECRETLEVELSND | secret | 0 | 0 | 255 | - | - | - | - | - | 0 | 11.025 |
| | Secret Level | | | | | | | | | | | |
| 184 | GENERIC_AMBIENCE8 | amb81b | 0 | 0 | 0 | - | - | - | \$ | - | 0 | 8 |
| | Far off alien ambience | | | | | | | | | | | |

| | | | | | | | | | | | | | |
|-----|--------------------|----------|---|---|-----|---|---|---|----|---|---|-------|--|
| 185 | GENERIC_AMBIENCE9 | roam98b | 0 | 0 | 0 | - | - | - | \$ | - | 0 | multi | Muffled voice |
| 186 | GENERIC_AMBIENCE10 | h2orush2 | 0 | 0 | 0 | - | - | - | \$ | - | 0 | 8 | Water flowing |
| 187 | GENERIC_AMBIENCE11 | projrun | 0 | 0 | 0 | - | - | - | \$ | - | 0 | 8 | Movie Projector |
| 188 | GENERIC_AMBIENCE12 | blank | 0 | 0 | 0 | - | - | - | - | - | 0 | 5.988 | A blank .voc file |
| 189 | GENERIC_AMBIENCE13 | pay02 | 0 | 0 | 255 | - | @ | # | - | - | 0 | 8 | Duke "Damn, those alien bastards are going to pay for shooting up my ride" |
| 190 | GENERIC_AMBIENCE14 | onlyon03 | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 | Duke "What, there's only one of you?" |
| 191 | + | | | | | | | | | | | | |
| 192 | GENERIC_AMBIENCE15 | rides09 | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 | Duke "I think I'll climb aboard" |
| 193 | GENERIC_AMBIENCE16 | doomed16 | 0 | 0 | 255 | - | - | # | - | - | 0 | multi | Duke "That's one doomed space marine" |
| 194 | FIRE_CRACKLE | fire09 | 0 | 0 | 254 | - | - | - | \$ | - | 0 | multi | Fire crackling |
| 195 | BONUS_SPEECH1 | letsrk03 | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 | Duke "Let's rock" |
| 196 | BONUS_SPEECH2 | ready2a | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 | Duke "Ready for action" |
| 197 | BONUS_SPEECH3 | ripem08 | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 | Duke "Rip 'ema new one" |
| 198 | PIG_CAPTURE_DUKE | !pig | 0 | 0 | 255 | - | @ | - | - | - | 0 | multi | "Got you now, you bastard, and we're gunna fry your arse" |
| 199 | BONUS_SPEECH4 | rockin02 | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 | Duke "Rockin' " |
| 200 | DUKE_LAND_HURT | pain39 | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 | Duke landing from a height and hurting himself |
| 201 | DUKE_HIT_STRIPPER1 | damnit04 | 0 | 0 | 255 | - | @ | # | - | - | 0 | 8 | Duke "Damn it" |
| 202 | DUKE_TIP1 | dance01 | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 | Duke "You wanna dance?" |
| 203 | DUKE_KILLED2 | damnit04 | 0 | 0 | 255 | - | @ | # | - | - | 0 | 8 | Duke "Damn it" |
| 204 | PRED_ROAM2 | roam58 | 0 | 0 | 3 | - | - | - | - | - | 0 | multi | Trooper roaming |

| | | | | | | | | | | | | |
|-----|--|-----------|------|-----|-----|---|---|---|----|---|------|-------|
| 205 | PIG_ROAM2 | roam67 | -200 | 400 | 3 | - | - | - | - | - | 0 | 8 |
| | Pig Cop roaming | | | | | | | | | | | |
| 206 | DUKE_GETWEAPON1 | cool01 | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 |
| | Duke "Cool" | | | | | | | | | | | |
| 207 | DUKE_SEARCH2 | whrsit05 | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 |
| | Duke "Whare is it?" | | | | | | | | | | | |
| 208 | DUKE_CRACK2 | COMEON02 | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 |
| | Duke "Come on" | | | | | | | | | | | |
| 209 | DUKE_SEARCH | pain87 | 0 | 0 | 2 | - | - | # | - | - | 0 | 8 |
| | Umph, Duke pressing on walls | | | | | | | | | | | |
| 210 | DUKE_GET | getitm19 | -64 | 64 | 255 | - | - | - | - | - | 0 | 8 |
| | Beep when Duke picks up an item | | | | | | | | | | | |
| 211 | DUKE_LONGTERM_PAIN | gasps07 | -192 | 0 | 255 | - | - | # | - | - | 0 | 8 |
| | Duke gasping | | | | | | | | | | | |
| 212 | MONITOR_ACTIVE | monitor | 0 | 0 | 0 | - | - | - | - | - | 0 | 8 |
| | Monitor beeping when changing camera | | | | | | | | | | | |
| 213 | NITEVISION_ONOFF | goggle12 | 0 | 0 | 0 | - | - | - | - | - | 0 | multi |
| | Turn on or turn off night vision googles | | | | | | | | | | | |
| 214 | DUKE_HIT_STRIPPER2 | damn03 | 0 | 0 | 255 | - | @ | # | - | - | 0 | 8 |
| | Duke "Damn" | | | | | | | | | | | |
| 215 | DUKE_CRACK_FIRST | knuckle | 0 | 0 | 3 | - | - | - | - | - | 0 | multi |
| | Duke cracking his knuckles | | | | | | | | | | | |
| 216 | DUKE_USEMEDKIT | ahh04 | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 |
| | Ahhh | | | | | | | | | | | |
| 217 | DUKE_TAKEPILLS | gulp01 | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 |
| | Gulp when Duke uses steroids | | | | | | | | | | | |
| 218 | DUKE_PISSRELIEF | ahmuch03 | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 |
| | Duke "Ahhh, much better" | | | | | | | | | | | |
| 219 | SELECT_WEAPON | WPNSSEL21 | 128 | 128 | 3 | - | - | - | - | - | 0 | 8 |
| | Changing weapon sound | | | | | | | | | | | |
| 220 | WATER_GURGLE | h2ogrgl2 | 0 | 0 | 1 | - | - | - | \$ | - | 9000 | multi |
| | Water trickling | | | | | | | | | | | |
| 221 | DUKE_GETWEAPON4 | wansom4a | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 |
| | Duke "Who wants some?" | | | | | | | | | | | |
| 222 | JIBBED_ACTOR1 | AMESS06 | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 |
| | Duke "What a mess" | | | | | | | | | | | |
| 223 | JIBBED_ACTOR2 | BITCHN04 | 0 | 0 | 255 | - | @ | # | - | - | 0 | 8 |
| | Duke "Bitchin' " | | | | | | | | | | | |
| 224 | JIBBED_ACTOR3 | HOLYCW01 | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 |
| | Duke "Holy Cow" | | | | | | | | | | | |

| | | | | | | | | |
|-----|---|----------|------|------|-----|-----------|------|--------|
| 245 | DUKE_SCREAM | DSCREM04 | 0 | 0 | 255 | - - - - - | 0 | 8 |
| | Duke scream when he falls from a great height | | | | | | | |
| 246 | SHRINKER_HIT | thud | 0 | 0 | 3 | - - - - - | 0 | multi |
| | Shrinker ray hit wall | | | | | | | |
| 247 | RATTY | mice3 | 0 | 0 | 0 | - - - - - | 0 | 11.025 |
| | Mice squeeking | | | | | | | |
| 248 | INTO_MENU | bulithit | 1024 | 1024 | 0 | - - - - - | 0 | 5.988 |
| | Enter menus | | | | | | | |
| 249 | BONUSMUSIC | bonus | 0 | 0 | 255 | - - - - % | 0 | 22.050 |
| | Music at end of level screen | | | | | | | |
| 250 | DUKE_BOOBY | BOOBY04 | 0 | 0 | 255 | - - # - - | 0 | 8 |
| | Duke "I should have known that those alien maggots booby trapped the sub" | | | | | | | |
| 251 | DUKE_TALKTOBOSSFALL | DIESOB03 | 0 | 0 | 255 | - @ # - - | 0 | multi |
| | Duke "Die you son of a bitch" | | | | | | | |
| 252 | DUKE_LOOKINTOMIRROR | lookin01 | 0 | 0 | 255 | - @ # - - | 0 | 8 |
| | Duke "Damn, I'm lookin' good" | | | | | | | |
| 253 | PIG_ROAM3 | pigrm | -200 | 400 | 3 | - - - - - | 0 | 8 |
| | Pig Cop roaming | | | | | | | |
| 254 | KILLME | killme | -128 | 0 | 0 | - - - - - | 0 | 11.025 |
| | Voice saying "Kill me" | | | | | | | |
| 255 | DRON_JETSND | ENGHUM | 1300 | 1300 | 0 | - - - - - | 0 | multi |
| | Engine humming | | | | | | | |
| 256 | SPACE_DOOR1 | hydro22 | 0 | 0 | 0 | - - - - - | 8192 | 11.025 |
| | Space door/elevator open/up | | | | | | | |
| 257 | SPACE_DOOR2 | hydro24 | 0 | 0 | 0 | - - - - - | 0 | 11.025 |
| | Space door/elevator close/down | | | | | | | |
| 258 | SPACE_DOOR3 | hydro27 | 0 | 0 | 0 | - - - - - | 8192 | 11.025 |
| | Space door/elevator open/up with stop | | | | | | | |
| 259 | SPACE_DOOR4 | hydro34 | 0 | 0 | 0 | - - - - - | 0 | 11.025 |
| | Space door open/close | | | | | | | |
| 260 | SPACE_DOOR5 | hydro40 | 0 | 0 | 0 | - - - - - | 0 | 11.025 |
| | Pressure release | | | | | | | |
| 261 | ALIEN_ELEVATOR1 | hydro43 | 0 | 0 | 0 | - - - - - | 0 | 11.025 |
| | Noisy elevator | | | | | | | |
| 262 | VAULT_DOOR | vault04 | 0 | 0 | 0 | - - - - - | 0 | 11.025 |
| | Slow moving door | | | | | | | |
| 263 | JIBBED_ACTOR13 | LETGOD01 | 0 | 0 | 255 | - - # - - | 0 | 11.025 |
| | Duke "Let God sort 'em out" | | | | | | | |
| 264 | DUKE_GETWEAPON6 | HAIL01 | 0 | 0 | 255 | - - # - - | 0 | 11.025 |
| | Duke "Hail to the King baby" | | | | | | | |

| | | | | | | | | | | | | | |
|-----|---------------------|----------|---|---|-----|---|---|---|----|---|---|--------|---|
| 265 | JIBBED_ACTOR8 | BLOWIT01 | 0 | 0 | 255 | - | @ | # | - | - | 0 | 11.025 | Duke "Blow it out your arse" |
| 266 | JIBBED_ACTOR9 | EATSHT01 | 0 | 0 | 255 | - | @ | # | - | - | 0 | 11.025 | Duke "Eat shit and die" |
| 267 | JIBBED_ACTOR10 | FACE01 | 0 | 0 | 255 | - | @ | # | - | - | 0 | 11.025 | Duke "Your face, your arse, what's the difference?" |
| 268 | JIBBED_ACTOR11 | INHELL01 | 0 | 0 | 255 | - | @ | # | - | - | 0 | 11.025 | Duke "See you in hell" |
| 269 | JIBBED_ACTOR12 | SUKIT01 | 0 | 0 | 255 | - | - | # | - | - | 0 | 11.025 | Duke "Suck it down" |
| 270 | DUKE_KILLED4 | dscrem18 | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 | Duke strained scream |
| 271 | DUKE_KILLED5 | pisses01 | 0 | 0 | 255 | - | @ | # | - | - | 0 | 11.025 | Duke "This really pisses me off" |
| 272 | ALIEN_SWITCH1 | aswtch23 | 0 | 0 | 0 | - | - | - | - | - | 0 | 11.025 | Splat switch |
| 273 | DUKE_STEPONFECES | happen01 | 0 | 0 | 0 | - | @ | # | - | - | 0 | 11.025 | Duke "Shit happens" |
| 274 | DUKE_LONGTERM_PAIN2 | dscrem15 | 0 | 0 | 255 | - | - | # | - | - | 0 | 8 | Medium length scream |
| 275 | DUKE_LONGTERM_PAIN3 | dscrem16 | 0 | 0 | 255 | - | - | # | - | - | 0 | 11.025 | Short scream |
| 276 | DUKE_LONGTERM_PAIN4 | dscrem17 | 0 | 0 | 255 | - | - | # | - | - | 0 | 11.025 | Even shorter scream |
| 277 | COMPANB2 | CTRLRM25 | 0 | 0 | 0 | - | - | - | \$ | - | 0 | 11.025 | Oscillating alien sound |
| 278 | KTIT | ktitx | 0 | 0 | 254 | - | - | - | \$ | - | 0 | 8 | Duke "This is KTIT, K tit. Bringing you the breast, I mean the best, tunes in town" |
| 279 | HELICOP_IDLE | hlidle03 | 0 | 0 | 255 | - | - | - | \$ | % | 0 | 5.988 | Helicopter blades spinning |
| 280 | STEPNIT | LIZSHIT3 | 0 | 0 | 254 | - | @ | # | - | - | 0 | 11.025 | Splat as Duke steps in it |
| 281 | SPACE_AMBIENCE1 | monolith | 0 | 0 | 0 | ! | - | - | - | - | 0 | 11.025 | Wierd alien ambience |
| 282 | SPACE_AMBIENCE2 | hydro50 | 0 | 0 | 0 | ! | - | - | - | - | 0 | 11.025 | Grinding moving sound |
| 283 | SLIM_HATCH | slhtch01 | 0 | 0 | 3 | - | - | - | - | - | 0 | 11.025 | Green slime hatching |

[illegible]

APPENDIX C

CON file Statement Summary

I don't know if Todd Replogle ever named the language used in the CON files, so I'll name it *DukeC* (pronounced "Dukesee"). What follows is a complete list of all the DukeC statements and their syntax.

Identifiers in [] are optional.

A statement block is defined as

```
{ statement
  statement ...}
```

COMPLETE LIST OF *DUKE C*

`action <action#> [<first frame>] [<#frames>] [<frameskip>] [<dir>] [<tempo>]`
defines a sprite frame sequence

`<action#>` : identifies the action. Usually a #define'd word

`<firstframe>`: offset from the starting frame of the actor

`<#frames>` : number of frames between each frame in the sequence

`<dir>` : 1 or -1. Allows frame sequences to go backwards

`<tempo>` : relative speed of each frame

`actor <actor#> [hp] [action] [ai] [state/code] enda` defines an actor

`<actor#>` : identifies the actor. Equals the first sprite of this actor.

`[hp]` : number of hit points the actor has. When 0, actor "dies"

`[action]` : the first action the actor performs

[ai] : the first ai the actor performs

[state/code]: either a call to a defined state, or a regular DukeC code block

enda : ends the actor block

addammo <weapon> <amount> Gives player <amount> units of ammo for <weapon>.

addkills <#> Adds <#> to the number of kills the player has made. Can be -1 for actors that respawn.

addinVENTORY <object> <amount>

<object> : #defined in DEFS.CON as GET_?????

<amount> : units of that object to give. Usually also pre-#defined

ex: addinventory GET_STEROIDS STEROID_AMOUNT

addphealth <#> adds <#> to player's current health. Can be negative to take health away.

addweapon <weapon> <#> Gives player weapon type <weapon>, with <#> units of ammo.

ai <ai#> <action> <speed> <basic ai> Defines an ai construct.

<ai#> : ai identifier

<action> : action (sprite sequence) when performing this ai

<speed> : movement of the actor when performing this ai

<basic ai> : one of "getv", "geth", "faceplayer", etc. defined in DEFS.CON

break Drops out of the current block of code, bypassing any other code in that block.

cactor <actor#> "call actor" This construct is used within an actor definition. It allows you to create an actor that uses the DukeC code from actor <actor#>. It is often used for creating creatures that start off in different action sequences than the standard actor Ex. This is the definition of actor PIGCOPDIVE, which defines the PigCop so that he starts off lying on the ground ready to shoot. The statement "cactor PIGCOP" is used so that this actor will use the actor code from the standard PIGCOP actor.

```
actor PIGCOPDIVE
    PIGCOPSTRENGTH
    ai AIPIGDIVING
    action APIGDIVESHOOT
    cactor PIGCOP
enda
```

cstat <#> Change the actor status to value #. cStat is a 16 bit integer, with each bit having a certain function. The function of each bit is below.

bit 0 (value 1): Make actor solid. Used w/ bit 7

bit 1 (value 2): make actor translucence

bit 2 (value 4): x-flip actor

bit 3 (value 8): y-flip actor

bits 5-4: 00 = sprite will always face actor (except mulit-frame monster sprites)

(value 16) 01 = sprite will face a single direction, usually along a wall.

Often used for posters, wall hangings, etc.

(value 32) 10 = sprite will face a single direction, parallel to the ceiling or floor.

bit 6 (value 64): One-sided sprite

bit 7 (value 128): Center sprite at actual center. Default is to center along the bottom

bit 8 (value 256): Make actor solid, used with bit 0

bit 9 (value 512): Translucence reversing, 0 = normal

bit 10 (value 1024): Unused

bit 11 (value 2048): Unused

bit 12 (value 4096): Unused

bit 13 (value 8192): Unused

bit 14 (value 16384): Unused

bit 15 (value 32768): Make sprite invisible

To achieve a desired effect, add up the desired bits' 'value' column.

For example, to make a sprite invisible, use cstat 32768. To make it visible again, use bits 0 and 8, which would be cstat 257.

cstat 0 is used to clear the solid bits, which allows the player to pass through the actor. This is usually set when the actor is killed, so that the player can walk through the corpse sprite.

debris <debris#> <amount> create <amount> pieces of sprite <debris#>, flying around. Usually used with the SCRAPn sprites. It might make an interesting effect to use a different sprite.

endoflevel <#> Ends the level. Usually used when player kills a boss-type monster. <#> is the time to pause before ending the level.

else Used with the if??? constructs below. Used to execute a block of code when a certain condition is NOT true.

fall Drops the actor to ground level. Used as the first line for many actors, this statement makes sure objects designed unintentionally as floating above ground level drop correctly.

globalsound <sound#> plays <sound#> 1 time everywhere in the level.

guts <guts#> <amount> Creates <amount> pieces of sprite <guts#>, flying around. Possibly the same internally as the debris statement, but used for body parts sprites.

hitradius <radius> <strength1> <strength2> <strength3> <strength4> Generates an explosion of <radius> wide. Damage taken by every action within this radius ranges from <strength1> at the outer edge to <strength4> at the center.

NOTE: All of the following statements are known collectively as if??? statements.

They all have the same basic syntax:

if??? [modifiers] {statement block1} [else {statement block2}] All of the functions test some condition to be true or false. If the condition is true, {statement

block1} is executed. If it is not true, and the else clause exists, then {statement block2} is executed. Note that if the condition is false and the else clause is NOT present, then NO clause is executed. Since all of the "if???" statements have the same syntax, the following descriptions will only explain what condition makes each statement true.

ifactornotstayput The "stayput" actor is used to define an actor that won't leave his home sector. This actor is used by finding the normal actor # in DEFS.CON and adding 1 to the value. This condition is true if the given actor is *not* of the "stayput" variety.

ifai <ai#> True if the actor is currently performing ai <ai#>.

ifaction <action#> True if actor is currently executing action sequence <action#>

ifactioncount <#> For each action begin executed, an internal counter is kept. It is automatically reset to 0 each time the actor changes actions. This statement is true if the internal counter has reached value <#>.

ifbulletnear True if a projectile is heading for the actor. Allows actor to perform a *dodgebullet* basic ai.

ifcansee True if current actor can see a player.

ifcanseetarget True if a player can see this actor. Used for inventory items.

ifcanshoottarget True if actor can currently shoot its current target: ie: the lines of sight are clear.

ifceilingdistl <#>

ifceilingdistg <#> True if the distance from the current actor to the ceiling is less than/greater than <#>

ifcount <#> True if internal counter is greater than <#>. Unknown exactly how this differs from ifactioncount

ifdead True if actor has 0 hit points.

iffloordistl <#>

iffloordistg <#> True if the distance from the current actor to the floor is less/greater than <#>.

ifgapz1 <#> True if the current sector height (ceiling to floor) is less than <#>

ifhitspace True if player hit the spacebar. For a great trick, place the line "if hitspace {spawn RPGAMMO}" immediately after the actor definition line for APLAYER in GAME.CON. You will create a box of RPG ammunition every time you hit the space bar!

ifhitweapon True if the actor was hit by a weapon.

ifinwater True if actor is under water.

ifmove <#> True if actor is moving at <#> speed.

ifnotmoving True if the current actor is not moving.

ifonwater True if actor is floating on water.

ifp <#> True if player is currently in one of the following states, defined in DEFS.CON

```
define pstanding 1
define pwalking 2
define prunning 4
define pducking 8
```



```

define pfalling 16
define pjumping 32
define phigher 64
define pwalkingback 128
define prunningback 256
define pkicking 512
define pshrunk 1024
define pjetpack 2048
define ponsteroids 4096
define ponground 8192
define palive 16384
define pdead 32768
define pfacing 65536

```

```

example:  ifp pjetpack { do stuff } else { do other stuff }

```

ifpdistl <#>

ifpdistg <#> True if player distance from actor is less/greater than <#>

ifphealthl <#>

ifphealthg <#> True if player health is less/greater than <#>

ifpinventory <item> <amount> True if player has inventory item <item> with less than <amount> units left.

ifplayersl <#>

ifplayersg <#> True if current network game has less/greater than <#> players

ifrespawn True if monster can respawn

ifrnd <#> True if a random number between 0 and 255 is less than <#>.

ifspawnedby <actor#> True if current actor was spawned by <actor#>.

ifspritepal <#> True if current actor's sprite is using palette <#>.

ifsquished True if the current actor is squished.

ifwasweapon <weapon#> True if weapon actor was just hit with was weapon <weapon#>.

<weapon#> can be one of:

RPG

THROWFLAME

SHRINKSPARK

RADIUSEXPLOSION

FIREEXT

KNEE

FIRELASER

COOLEXPLOSION1

SHOTSPARK1

MORTER

SHOTGUN

FIRSTGUN

include <filename> Compiles <filename>.

killit Removes current actor from level

lotsofglass <#> Spawns <#> units of "glass" shards. Used when a frozen actor is hit, causing him to break up.

money <#> Spawns <#> dollar bills. Used when a stripper is shot, or the money jar in the bar on level E1L2.

move <move#> <v#> [<h#>] Used to define a speed named <move#>. The speed is <v#>

units along the x-y plane and <h#> units in the z direction (for floating/flying objects). The <move#> can then be used in ai statements.

operate Allows the actor to open a closed door if nearby. Actors can not operate other effects like switches.

palfrom <p1> <p2> <wait> Rotates the screen palette from <p1> to <p2>, using a delay of <wait>. Used for a "flash of red" when Duke walks through fire, for example.

pstomp Used only in state genericshrunkcode, makes player look down and squish an actor. Used when monster is shrunk to exterminate him.

quote <#> Displays quote <#> at the top of the screen. Quotes are defined in USER.CON with the "definequote" statement.

resetactioncount Sets the current internal action count for this action to zero. This is done automatically if an action changes.

resetcount Sets the current internal count used by "ifcount" to zero.

shoot <weapon#> Causes the actor to shoot. Valid values for <weapon#> can be found above under "ifwasweapon" statement.

sizeto <x#> <y#> Changes the size of an actor to <#x>/256 horizontal and <#y>/256 vertical. Used when an actor teleports (only the x direction shrinks), and of course, when an actor is hit with a shrink gun.

sound <sound#> Plays sound effect <sound#>. <Sound#> values are defined in DEFS.CON. Sounds can be played multiple times in succession and DN3D will repeat each sound as long as there is an open channel to play it. This allows the same sound to be played more than once at the same time, creating an echo effect.

soundonce <sound#> Plays the given sound only if it is not already playing.

spawn <actor#> Create a new <actor#>.

spritepal <#> Changes the sprite palette to <#>. For example, palette 1 is a mostly blue palette, used for freezing objects.

state <state#> <statement block> ends Used for define code block <state#>. The code block must be defined before calling it.

state <state#> Calls code block <state#>

strength <#> Changes an actor's hit points to <#>.

useractor <actortype> <actor#> [hp] [action] [ai] [state/code] enda defines a new user-created actor

<actor#> : identifies the actor. Equals the first sprite of this actor.

<actortype> : either enemy, enemystayput, or notenemy.

[hp] : number of hit points the actor has. When 0, actor "dies"

[action] : the first action the actor performs

[ai] : the first ai the actor performs

[state/code]: either a call to a defined state, or a regular DukeC code block

enda : ends the actor block

wackplayer Knocks player's view around for a few seconds.



WHAT'S ON THE CD

Shareware versions of *Duke Nukem 3D* and six other Apogee Software and 3D Realms Entertainment popular offerings—*Raptor*, *Realms of Chaos*, *Rise of the Triad*, *Terminal Velocity*, *Wacky Wheels*, and *Xenophage*—can be found in the Apogee directory. Each game comes with an automated installation program. See *Readme.txt* for details.

If you've got piles of *DOOM*, *Heretic*, and *Hexen* levels sitting around on your hard drive (doesn't everyone?), you'll probably want to start by taking a look at the WAD2DUKE program. This can be found in the \WAD2DUKE directory on the CD. This program will take these levels and convert them into levels you can play in *Duke Nukem 3D*. See chapter 13 for instructions on how to perform this conversion.

The \WAD2DUKE\DEMO directory contains the sample WAD file that's converted into a *Duke Nukem 3D* level. There are three copies of this level: the original WAD file (LOSTE1M3.WAD), the level immediately after conversion (LOSTE1M3.MAP), and the same level after fixing it up a bit (LST2E1M3.MAP). This is the same level that's shown in chapter 13 as an example.

There are some good sample CON files to play around with in the CONFILES\ directory. PIGPORT.CON is a replacement GAME.CON file that allows the Pig Cops to teleport around after they get shot, which can be quite a nuisance to a player. GAME14.CON shows you how to create new actor code for an all new monster, which is possible in the version 1.4 of *Duke Nukem 3D*.

If you have an interest in programming, you'll find a sample program in the \DUKE-VIEW directory that will load and display a *Duke Nukem 3D* MAP file. The program is written in Delphi 1.0. This project could be used to get you started in writing your own *Duke Nukem 3D* level editor.

The EDITART\ directory contains some artwork that can be used to create some new objects for your *Duke Nukem 3D* levels: a three-frame piston, and a vase. There's also a DUKE3D.PAL file that contains the palette which comprises the *Duke Nukem 3D* graphics. This file can be used in Paint Shop Pro for converting artwork to the *Duke Nukem 3D* palette.

You will find two of the best available online compilations of *Duke Nukem 3D* level creation issues in the FAQ directory. The first is from Klaus Breuer, and is located in the \FAQ\KLAUS directory. Klaus's FAQ also comes with a demo map, named MUSEUM.MAP, that highlights all of the effects and objects at your disposal for level building in a single level. The second FAQ is compiled by Steffen "Duke Addict" Itterheim, and can be found in the \FAQ\DADDICT directory. Steffen is quite a *Duke Nukem 3D* enthusiast, and also included a few of his own converted levels, as well as his special sprites tutorial.

Finally, in the \MAPS directory, you will find a few dozen levels created by some of the most dedicated *Duke Nukem 3D* players around. There's a nice mix of both single player and multi-player maps here, so no matter what your favorite mode of *Duke Nukem 3D* play, there's bound to be something here that you'll find interesting.

OFFICIAL

DUKE NUKEM® 3D

LEVEL DESIGN HANDBOOK

50 New Duke Nukem
3D Levels

EXCLUSIVE DOOM/
Heretic/Hexen WAD
Conversion Utility

EXCLUSIVE Level
Viewer/Editor

Shareware Versions of
3D Realms Games and
Duke Nukem 3D

Sample Art and
Con Files



SYBEX
1869-0



© SYBEX, Inc., 1996. Files on this disk are
copyrighted by their respective owners
and have been so noted.

THE DUKE NUKEM 3D LEVEL DESIGN HANDBOOK

Design the **BEST**
Duke Nukem 3D
Levels

Discover the
Insider Secrets of
Level Design

Get Tips from
The Levelord and
Allen S. Blum III,
Designers of the
Original Duke 3D
Levels

The Official Duke Nukem 3D Level Design Handbook is the bible for creating totally new Duke Nukem 3D levels. Written in cooperation with 3D Realms and the maker of the Build engine, this official handbook includes an exclusive DOOM WAD conversion program and valuable insider

information—including tips and tricks from the actual level designers of the original Duke Nukem 3D game.

Graduate from Duke “fan” to Duke “creator.”

3D Realms has given you the Tools; this book gives you the Knowledge.

- Master “Build,” the 3D level editor
- Create Sector-Effectors
- Use textures, shading, and palettes like Duke da Vinci
- Fill the world with weapons, medkits, and other sprites

BECOME LORD & MASTER OF THE 3D UNIVERSE!

- Create your own **NASTY** monsters
- Populate your levels with awesome objects
- Learn the secrets of teleportation, subways, mirrors, and other constructs

Make the best levels ever. Discover the differences between single- and multi-player level design. Employ suspense, surprise, risk, reward, and other professional level design tricks. Change the game’s artwork with EditArt—or modify CON files to actually change the way the game plays.

Get INSIDE information! Richard Bailey Gray (a.k.a. The Levelord) and Allen S. Blum III, designers of the original Duke Nukem 3D levels, show you how to create levels that people will clamor to play again and again.

50 NEW LEVELS!



CONVERT DOOM/HERETIC LEVELS INTO DUKE NUKEM 3D LEVELS!

This CD-ROM includes a *working* conversion utility that converts all your DOOM, Heretic, and Hexen levels into Duke Nukem 3D levels. With this *exclusive* conversion program—available *only* to our readers—you can add thousands of levels of new Duke Nukem adventures. You’ll also find 50 outstanding, ready-to-run levels, designed by dedicated Duke-heads nationwide. In addition, the CD features shareware versions of Duke Nukem 3D and other popular Apogee and 3D Realms games; tips for modifying the game’s parameters using CON files; and sample art you can import into your own levels using the EditArt utility.

ABOUT THE AUTHOR

matt tagliaferri is a professional programmer and software engineer. As creator of DOOMCAD, the widely acclaimed DOOM level editor, he is well-known in the online gaming community.



COMPUTER BOOK SHELF CATEGORY

PC: Games

ISBN 0-7821-1869-0 U.S. \$24.99



Made with love by

RETROMAGS

Our goal is to preserve classic video game magazines so that they are not lost permanently.

People interested in helping out in any capacity, please visit us at retromags.com.

No profit is made from these scans, nor do we offer anything available from the publishers themselves.

If you come across anyone selling releases from this site, please do not support them and do let us know.

Thank you!